Hans Petter Langtangen[*], Anders Logg[†]

Writing Advanced PDE Solvers in Hours - The FEniCS Tutorial Volume II

Jun 2, 2016

Springer

[*]Email: hpl@simula.no. Center for Biomedical Computing, Simula Research Laboratory and Department of Informatics, University of Oslo.

[†]Email: logg@chalmers.se. Department of Mathematics, Chalmers University of Technology and Center for Biomedical Computing, Simula Research Laboratory.

Contents

1 Developing a more advanced heat equation solver 1
1.1 A flexible and efficient solver 1
1.1.1 Numerical method 1
1.1.2 Algorithm 3
1.1.3 The solver function 4
1.1.4 Marking the boundary 5
1.1.5 Implementation of the variational formulation 7
1.1.6 The time loop 8
1.1.7 Verification 10
1.1.8 Debugging of FEniCS programs 14
1.1.9 Avoiding all assembly 17
1.1.10 Lumped mass matrix 21
1.2 A welding example with post processing and animation 27
1.2.1 Post processing data and saving to file 27
1.2.2 Heat transfer due to a moving welding source 28
1.2.3 Scaling of the welding problem 29
1.2.4 A function-based solver 31

2 PDE solver design and coding practices 35
2.1 Refactoring a Poisson solver in terms of classes 35
2.1.1 The solver class 36
2.1.2 A problem class 39
2.1.3 A more complicated problem class 39
2.2 Refactoring a heat equation solver 42
2.2.1 Mathematical problem 43
2.2.2 Superclass for problems 45
2.2.3 A specific problem class 46
2.2.4 The solver class 48
2.3 Applications to heat conduction 50
2.3.1 Thermal boundary layer 50
2.3.2 Extension to a heterogeneous medium 55

2.3.3 Oscillating boundary temperature 58

3 Implementing solvers for nonlinear PDEs 63
3.1 The built-in automated Newton solver 64
3.1.1 Computing the Jacobian 64
3.1.2 Setting solver parameters 66
3.1.3 FEniCS implementation 67
3.2 Manual implementation of solution algorithms 68
3.2.1 Picard iteration 68
3.2.2 A Newton method at the algebraic level 70
3.2.3 A Newton method at the PDE level 73
3.2.4 Implementations with functions and classes 75

4 Mixed finite element programming 77

5 High-performance computing 79

6 Multi-physics in multi-domains 81

7 More old stuff 83
7.1 Solving Neumann problems 83
7.2 Transforming hypercube domains 83
7.2.1 Coordinate stretching 83
7.2.2 Rectangle to hollow circle mapping 84
7.3 Encyclopedia stuff 86
7.3.1 Glossary 86
7.3.2 Overview of objects and functions 87
7.3.3 Handy methods in key FEniCS objects 88
7.3.4 Using a backend-specific solver 89

8 Troubleshooting 93
8.1 Compilation Problems 93
8.1.1 Problems with the Instant cache 93
8.1.2 Syntax errors in expressions 94
8.1.3 An integral without a domain is now illegal 94
8.1.4 Problems with SymPy and `diff` expressions 95
8.1.5 Problems in the solve step 95
8.1.6 Unable to convert object to a UFL form 95
8.1.7 UFL reports that a numpy array cannot be converted to any UFL type 96
8.1.8 All programs fail to compile 96
8.2 Problems with Expression Objects 96
8.2.1 There seems to be some bug in an Expression object . . 96
8.2.2 Segmentation fault when using an Expression object . . 97
8.3 Other Problems 97
8.3.1 Very strange error message involving a `mesh` variable . . 97

8.3.2 The plot disappears quickly from the screen 97
8.3.3 Only parts of the program are executed 97
8.3.4 Error in the definition of the boundary 98
8.3.5 The solver in a nonlinear problems does not converge . 98
8.4 How To Debug a FEniCS Program? 98
8.5 To-do list 99
8.5.1 AL list 99
8.5.2 HPL list 101
8.5.3 HPL questions 102

References 105

Index 107

Chapter 1
Developing a more advanced heat equation solver

This chapter is devoted to some important issues when solving time-dependent problems with FEniCS: avoiding unnecessary assembly, dealing with time-dependent `Expression` objects, debugging the coding of variational forms, lumping mass matrices, saving results to file, and making animations. We handle these topics through a welding problem and address all aspects of code development, from scaling of the physical problem via debugging to constructing unit tests and sharing FEniCS best practices.

The PDE to be addressed is the heat equation

$$\varrho c \frac{\partial u}{\partial t} = \nabla \cdot (p \nabla u) + f,$$

with initial condition $u = I$ and various types of Dirichlet, Neumann, and Robin conditions. The primary unknown is supposed to represent the temperature, and the PDE governs heat transport in a solid heterogeneous material. The physical parameters, which may vary in space, are the density of the medium, ϱ, the heat capacity, c, and the heat conduction coefficient, p, while f is a heat source.

A very simple FEniCS program for a diffusion equation was introduced in Section 3.1 in [2]. You should be familiar with that code prior to reading the present chapter as the code to be presented has many more advanced features.

1.1 A flexible and efficient solver

1.1.1 Numerical method

Let us use a θ rule for discretizing the problem in time. Given

$$\frac{\partial u}{\partial t} = \mathcal{G}(u) + f,$$

where $\mathcal{G}$ is some differential operator and f some source term, the θ rule reads

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta\mathcal{G}(u^{n+1}) + (1-\theta)\mathcal{G}(u^n) + f^{n+\theta}, \tag{1.1}$$

or

$$\frac{u^{n+1} - u^n}{\Delta t} = \mathcal{G}(\theta u^{n+1} + (1-\theta)u^n) + f(t_{n+\theta}). \tag{1.2}$$

These equations are equal only if $\mathcal{G}$ is a linear operator. The notation $f^{n+\theta}$ means a weighted average of f at time points t_n and $t_n n+1$: $\theta f^{n+1} + (1-\theta)f^n$, while $f(t_{n+\theta})$ means f evaluated at the time point $\theta t_{n+1} + (1-\theta)t_n$.

The nice feature of the θ rule is that it reproduces three widely used discretization methods in time: $\theta = 0$ gives a classical Forward Euler scheme, $\theta = 1$ gives a Backward Euler scheme, and $\theta = \frac{1}{2}$ gives a Crank-Nicolson (or midpoint/centered) scheme. The latter is theoretically the most accurate, but suffers from non-physical oscillations of high-frequency components of the solution, so many applications may demand the more stable Backward Euler scheme (or a more accurate backward difference formula utilizing a third time level).

The corresponding variational formulation for u^{n+1} is derived by multiplying the time-discrete PDE (**??**) or (**??**) by a test function $v \in \hat{V}$ and integrating over the spatial domain Ω. Terms with second-order derivatives are integrated by parts. We can express the integration by parts as

$$\int_\Omega \mathcal{G}(u)v\,\mathrm{d}x = -\int_\Omega \mathcal{D}(u,v)\,\mathrm{d}x + \int_\Gamma \mathcal{B}(u,v)\,\mathrm{d}s\,.$$

Using (1.2), and introducing

$$U = \theta u^{n+1} + (1-\theta)u^n),$$

the variational formulation becomes

$$F = \int_\Omega \varrho c\frac{u^{n+1} - u^n}{\Delta t}v\,\mathrm{d}x + \int_\Omega \mathcal{D}(U,v)\,\mathrm{d}x - \int_\Omega f(\theta t_{n+\theta})v\,\mathrm{d}x + \quad \int_\Gamma \mathcal{B}(U,v)\,\mathrm{d}s \tag{1.3}$$

Note that we have inserted a factor ϱc in the time-derivative term since our PDE has this factor. Also note that all terms are evaluated at the time point $t_{n+\theta}$.

We introduce a general initial condition

$$u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}) \text{ in } \Omega\,.$$

As boundary conditions, we assume either Dirichlet conditions on the entire boundary or a Robin condition

$$-p\frac{\partial u}{\partial n} = r(u-s),$$

where r is a heat transfer coefficient and s is the surrounding temperature. Note that insulated boundaries are modeled by $r = 0$. For the present physical problem we have

$$\mathcal{D}(u,v) = \nabla\cdot(p\nabla u^n),$$
$$\mathcal{B}(u,v) = r(u-s)v\,.$$

Normally, in finite element programs, we would need to break up the variational formulation (1.3) into a bilinear and a linear part, but in FEniCS we can just use `lhs(F)` and `rhs(F)` for such calculations, which is very convenient from a user's point of view. The final version of the variational formulation to be coded reads

$$F = \int_\Omega (\varrho c\frac{u^{n+1}-u^n}{\Delta t}v + p\nabla U\cdot\nabla v - f(\theta t_{n+\theta})v)\,\mathrm{d}x + \int_\Gamma r(U-s)v\,\mathrm{d}s \quad (1.4)$$

If s varies with time, we need to evaluate it as $s(t_{n+\theta})$.

1.1.2 Algorithm

Let us express the solution procedure in algorithmic form, writing u for the unknown spatial function at the new time level (u^n) and u_1 for the spatial solution at one earlier time level (u^{n-1}):

- mark boundary segments for prescribing boundary conditions
- let initial condition u^n interpolate I or be the projection of I
- define F
- ask FEniCS to recognize $a(u,v)$ and $L(v)$
- assemble matrix A from $a(u,v)$ if A is time independent
- assign some stopping time T
- $t = \Delta t$
- while $t \leq T$
  - update time-dependent objects with new time
  - assemble matrix A from $a(u,v)$ if A is time dependent
  - assemble vector b from L
  - apply essential boundary conditions
  - solve linear system

- $t \leftarrow t + \Delta t$
- $u^n \leftarrow u$ (be ready for next step)

Our time-dependent heat equation gives rise to a linear system with coefficient matrix A and right-hand side b at every time level. When ϱ, c, p, and r do not depend on time, and Δt is constant, A is constant, and it suffices to assemble the matrix once – before the time loop. To be able to do this, we need to explicitly create matrices and vectors from variational formulations, using the `assemble` function.

The code features a lot of changes from the `ft03_heat.py` program. We shall go through each part of the above algorithm.

1.1.3 The solver function

Instead of a flat program, we wrap the solver in a function:

```
def solver(
    rho, c, p, f, r, s, u0, T, L,       # physical parameters
    dt, divisions, degree=1, theta=1,   # numerical parameters
    user_action=None,                   # callback function
    u0_project=False,                   # project/interpolate u0
    BC='Dirichlet',                     # interpretation of r
    A_is_const=False,                   # is A time independent?
    debug=False):
```

We assume that the domain is an interval, rectangle, or box, with dimensions given by the list `L` and where `divisions` specifies the number of cells in each spatial direction. Alternatively, we could demand that a ready-made mesh is provided by the calling code, but we take the opportunity here to illustrate once again the setting of different boundary conditions at different parts of the boundary.

Boundary condition conventions. A convention is introduced for the boundary conditions: if `BC == 'Dirichlet'`, the variable `r` is a list with Dirichlet values (`Constant` or `Expression` objects) for each side of the domain. Side 0 means $x = 0$, 1 is $x = 1$, 2 is $y = 0$, 3 is $y = 1$, 4 is $z = 0$, and 5 is $z = 1$. If `BC == 'Robin'`, `r[i]` holds the heat transfer coefficient for boundary side `i`. (The variable `s`, related to the Robin condition, has no meaning if `BC == 'Dirichlet'`).

Checking input data. It is wise to start the function with checking the values of some of the input parameters. Python's `assert` function is ideal for quick writing of tests, at the cost of cryptic error messages for less experienced programmers. When we have some test expressed as a boolean condition `cond`, we can just write `assign cond`. The statement is silent if `cond` is `True`, otherwise an exception is raised and the program is aborted.

```
assert len(divisions) == len(L)
d = len(L)  # no of space dimensions
assert len(r) == 2*d
for obj in p, f, s:
    assert isinstance(obj, (Expression, Constant))
if user_action is not None: assert callable(user_action)
```

Creating the mesh. For now we restrict the geometry to a hypercube (but the `solver` function can compute on any type of domain and mesh). We use the recipe from Section 6.1.5 in [2].

```
if d == 1:
    mesh = IntervalMesh(divisions[0], 0, L[0])
elif d == 2:
    mesh = RectangleMesh(Point(0,0), Point(*L), *divisions)
elif d == 3:
    mesh = BoxMesh(Point(0,0), Point(*L), *divisions)
V = FunctionSpace(mesh, 'P', degree)
```

Note that `*L` for a list or tuple variable `L` in a function call is the same as sending the elements as separate arguments `L[0], L[1], ..., L[len(L)-1]`.

1.1.4 Marking the boundary

We need to mark each side of our hypercube domain since we can have Robin or Dirichlet conditions that differ on different sides. We could write something straightforward as

```
def mark_boundaries_in_rectangle(mesh, x0=0, x1=1, y0=0, y1=1):
    """
    Return mesh function FacetFunction with each side in a rectangle
    marked by boundary indicator 0, 1, 2, 3.
    Side 0 is x=x0, 1 is x=x1, 2 is y=y0, and 3 is y=y1.
    """
    tol = 1E-14

    class BoundaryX0(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary and near(x[0], x0, tol)

    class BoundaryX1(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary and near(x[0], x1, tol)

    class BoundaryY0(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary and near(x[1], y0, tol)

    class BoundaryY1(SubDomain):
        def inside(self, x, on_boundary):
```

```
            return on_boundary and near(x[1], y1, tol)

    # Mark boundaries
    boundary_parts = FacetFunction('uint', mesh)
    boundary_parts.set_all(9999)
    bx0 = BoundaryX0()
    bx1 = BoundaryX1()
    by0 = BoundaryY0()
    by1 = BoundaryY1()
    bx0.mark(boundary_parts, 0)
    bx1.mark(boundary_parts, 1)
    by0.mark(boundary_parts, 2)
    by1.mark(boundary_parts, 3)
    return boundary_parts
```

Unfortunately, this is quite tedious and repetitive code, and the code has to be repeated for a 1D interval and a 3D box-shaped domain. It is possible to write more general, compact code valid both for an interval, rectangle, or box:

```
def mark_boundaries_in_hypercube(
    mesh, d=2, x0=0, x1=1, y0=0, y1=1, z0=0, z1=1):
    """
    Return mesh function FacetFunction with each side in a hypercube
    in d dimensions. Sides are marked by indicators 0, 1, 2, ..., 6.
    Side 0 is x=x0, 1 is x=x1, 2 is y=y0, 3 is y=y1, and so on.
    """
    side_definitions = [
        'near(x[0], %(x0)s, tol)', 'near(x[0], %(x1)s, tol)',
        'near(x[1], %(y0)s, tol)', 'near(x[1], %(y1)s, tol)',
        'near(x[2], %(z0)s, tol)', 'near(x[2], %(z1)s, tol)']
    boundaries = [CompiledSubDomain(
        ('on_boundary && ' + side_definition) % vars(), tol=1E-14)
                  for side_definition in side_definitions[:2*d]]
    # Mark boundaries
    boundary_parts = FacetFunction('uint', mesh)
    boundary_parts.set_all(9999)
    for i in range(len(boundaries)):
        boundaries[i].mark(boundary_parts, i)
    return boundary_parts
```

The use of this function in the solver function goes as follows:

```
boundary_parts = mark_boundaries_in_hypercube(mesh, d)
ds =  Measure('ds', domain=mesh, subdomain_data=boundary_parts)

bcs = []
if BC == 'Dirichlet':
    for i in range(2*d):
        bcs.append(DirichletBC(V, r[i], boundary_parts, i))
```

Recall that we *either* have Dirichlet *or* Robin conditions at the entire boundary. This is just a convention that we, as developers of the `solver` function, has imposed. Later, in Chapter 2, we leave it to the user to define the mesh

and boundary conditions, which results in a very flexible solver, but a significant amount of problem-specific code on the user's side.

hpl 1: I'm really not sure whether it would be better to require a mesh variable from the user also here, along with some representations of the boundary conditons, as arguments for `solver` and put the setting of boundary conditions in the calling code. Now the examples with a *function* `solver` operates on hypercube geometries and deal with the partitioning of the boundary and setting of conditions, while the examples with a solver *class* just requires an API where the user supplies the mesh and info on boundary conditions. Maybe best with showing both ways. The bottom line is to explain something sufficiently general so the user does not copy the entire solver to change the boundary conditions and ends up with multiple solvers.

1.1.5 Implementation of the variational formulation

We shall now implement the variational formulation and demonstrate how that formulation can be coded in separate parts, utilizing Python variables and functions. Later, we can examine the individual parts in search for programming errors.

We start with implementing the initial condition:

```
u_n = project(u0, V) if u0_project else interpolate(u0, V)
u_n.rename('u', 'initial condition')
if user_action is not None:
    user_action(0, u_n, 0)
```

In the variational form, we make use of some convenient constructions like `U` as the θ weighted averaged of u in time and separate Python functions for various terms in the formulation:

```
u = TrialFunction(V)
v = TestFunction(V)

def D(u):
    return p*dot(grad(u), grad(v))*dx

def B(u, i):
    return r[i]*(u-s)*v*ds(i)

# In time loop: must set the t attribute in f, s, and r[i] to
# theta*t + (1-theta)*(t-dt) before evaluating the forms

U = theta*u + (1-theta)*u_n
F_M = rho*c*(u-u_n)/dt*v*dx
F_K = D(U)
F_f = f*v*dx
F = F_M + F_K - F_f
if BC == 'Robin':
```

```
    # Add cooling condition integrals from each side
    F_R = sum(B(U, i) for i in range(2*d))
    F += F_R
a, L = lhs(F), rhs(F)
```

We have with purpose split the expression for `F` into separate terms for easier debugging later, as this allows us to assemble terms independently and compare with hand calculations.

It remains to assemble the coefficient matrix, here once and for all before the time loop if A is constant throughout the simulations:

```
if A_is_const:
    A = assemble(a)
```

At each time level we must do a similar `b = assemble(L)`. However, with this construction, a new vector for `b` is allocated in memory in every pass of the time loop. It would be much more memory friendly to reuse the storage of the `b` we already have. This is easily accomplished by

```
b = assemble(L, tensor=b)
```

That is, we send in our previous `b`, which is then filled with new values and returned from `assemble`. Now there will be only a single memory allocation of the right-hand side vector. Before the time loop we must set `b = None` such that `b` is defined as a variable in the first call to `assemble` inside the time loop.

1.1.6 The time loop

The complete time loop goes as follows:

```
u = Function(V)   # the unknown at a new time level
u.rename('u', 'solution')
cpu_assemble = 0  # CPU time for assembling
timestep = 1
t = dt

while t <= T:
    # Evaluate f, s, r[i] for right t value
    t_m = theta*t + (1-theta)*(t-dt)
    if hasattr(f, 't'): f.t = t_m
    if hasattr(s, 't'): s.t = t_m
    for i in range(len(r)):
        if BC == 'Robin':
            if hasattr(r[i], 't'): r[i].t = t_m
        elif BC == 'Dirichlet':
            if hasattr(r[i], 't'): r[i].t = t
        else:
            raise ValueError('BC=%s' % BC)
    t0 = time.clock()  # measure CPU time of assemble part
```

```
    if not A_is_const:
        A = assemble(a)
    b = assemble(L, tensor=b)
    cpu_assemble += time.clock() - t0

    [bc.apply(A, b) for bc in bcs]
    solve(A, u.vector(), b)

    if user_action is not None:
        user_action(t, u, timestep)
    t += dt
    timestep += 1
    u_n.assign(u)
```

Make sure Expression objects have the right time value!

The first part of the loop, where we update `Expression` objects, is really key to get right and one of the most error-prone tasks for FEniCS programmers. All given formulas in the variational form are to be evaluated at the intermediate time point `t_m` ($t_{n+\theta}$). The variational forms can work with time-dependent `Expression` objects and evaluate the time variable in these objects when we require an `assemble` operation. Hence, at each time level, every `Expression` object that enters the variational formulation that is subject to `assemble` calls must have its right time value (or more precisely and general: all of its parameters must have the correct values). In addition, `Expression` objects related to Dirichlet values must contain the same time value as that of the unknown to be computed. In the present case, `f`, `s`, and `r[i]` enter the variational formulation at the weighted time $t_m = t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_{n+1}$, so this time value must be assigned to the `t` attribute in these objects. However, it may happen that one or more of the objects are `Constant` objects, or `Expression` objects without a time value, so a straight assignment `f.t = t_m` may fail. Therefore, we use `hasattr` to check that the object has a `t` attribute before trying to update the value.

The update of `r[i].t` depends on whether `r` is used for Dirichlet or Robin conditions. In the latter case, the r quantity is to be evaluated at the weighted time, `r[i].t = t_m`, while for a Dirichlet condition, `r[i].t` must reflect the same time level as the unknown we compute for, i.e., t_{n+1}, or the variable `t` in the time loop.

The rest of the statements in the time loop should be quite familiar. Note that `[bc.apply(A, b) for bc in bcs]` is a quick way of writing a for loop on one line (using Python's list comprehension syntax, but the resulting list is never used for anything, just the calls `bc.apply(A, b)` are important for incorporating the Dirichlet conditions at each boundary segment).

The complete `solver` function is found in the file `ft12_heat_func.py`. **hpl 2**: Recall to rerun utility script for renaming these files prior to refereeing and publishing.

1.1.7 Verification

The first implementation of a solver of the complexity above is likely to suffer from programming errors or mathematical misunderstandings. We must therefore carefully set up tests so that we know that the implementation works. As usual, we favor manufactured solutions that can be exactly reproduced by the numerical method. With variable coefficients and a lot of input data to adjust, the choice of manufactured solution must be flexible. We therefore feed some symbolic expression for $u(\boldsymbol{x},t)$ into a function `verify` and let this function compute the consistent source term and the coefficients `r[i]` in the Robin/Dirichlet conditions. Then `solver` is called with a callback function that asserts the error to be within machine precision for this problem, if the manufactured solution is without approximation errors.

We use SymPy to do the mathematics and the code generation utility in SymPy to translate the symbolic expressions to C++ code needed in FEniCS `Expression` objects.

```
def verify(
    manufactured_u,         # SymPy expression
    d=2,                    # no of space dimensions
    degree=1,               # degree of finite element polynomials
    BC='Robin',             # type of boundary condition
    N=16,                   # partitioning in each space direction
    theta=1,                # time discretization parameter
    expect_exact_sol=True,  # True: no approximation errors
    lumped_mass=False,      # True: lump mass matrix
    avoid_b_assembly=False, # True: construct b as matrix-vector products
    A_is_const=None,        # None: set True if BC is 'Dirichlet'
    debug=False,            # True: dump a lot of debugging info
    error_tol=1E-13):       # tolerance for exact numerical solution

    u = manufactured_u  # short form
    x, y, z, t = sym.symbols('x[0] x[1] x[2] t')

    if d == 1:  # 1D test problem
        p = 1
        s = 1
        rho = c = 1
        # Fit f, r[i]
        f = rho*c*sym.diff(u, t) - sym.diff(p*sym.diff(u, x), x)
        f = sym.simplify(f)
        # Boundary conditions: r = -p*(du/dn)/(u-s)
        r = [None]*(2*d)
        r[0] = (+p*sym.diff(u, x)/(u-s)).subs(x, 0)
```

```
        r[1] = (-p*sym.diff(u, x)/(u-s)).subs(x, 1)

    elif d == 2:  # 2D
        p = 1
        s = 2
        rho = c = 1
        f = rho*c*sym.diff(u, t) \
            - sym.diff(p*sym.diff(u, x), x) \
            - sym.diff(p*sym.diff(u, y), y)
        f = sym.simplify(f)           # fitted source term
        # For Robin boundary conditions: r = -p*(du/dn)/(u-s)
        r = [None]*(2*d)
        r[0] = (+p*sym.diff(u, x)/(u-s)).subs(x, 0)
        r[1] = (-p*sym.diff(u, x)/(u-s)).subs(x, 1)
        r[2] = (+p*sym.diff(u, y)/(u-s)).subs(y, 0)
        r[3] = (-p*sym.diff(u, y)/(u-s)).subs(y, 1)

    elif d == 3:  # 3D
        p = 1
        s = 2
        rho = c = 1
        f = rho*c*sym.diff(u, t) \
            - sym.diff(p*sym.diff(u, x), x) \
            - sym.diff(p*sym.diff(u, y), y) \
            - sym.diff(p*sym.diff(u, z), z)
        f = sym.simplify(f)           # fitted source term
        # For Robin boundary conditions: r = -p*(du/dn)/(u-s)
        r = [None]*(2*d)
        r[0] = (+p*sym.diff(u, x)/(u-s)).subs(x, 0)
        r[1] = (-p*sym.diff(u, x)/(u-s)).subs(x, 1)
        r[2] = (+p*sym.diff(u, y)/(u-s)).subs(y, 0)
        r[3] = (-p*sym.diff(u, y)/(u-s)).subs(y, 1)
        r[4] = (+p*sym.diff(u, z)/(u-s)).subs(z, 0)
        r[5] = (-p*sym.diff(u, z)/(u-s)).subs(z, 1)

    for i in range(len(r)):
        r[i] = sym.simplify(r[i])
    print('f:', f, 'r:', r)

    # Convert symbolic expressions to Expression or Constant
    s = Constant(s)
    rho = Constant(rho)
    c = Constant(c)
    f = Expression(sym.printing.ccode(f), t=0)
    p = Expression(sym.printing.ccode(p))
    u_exact = Expression(sym.printing.ccode(u), t=0)

    if BC == 'Dirichlet':
        for i in range(len(r)):
            r[i] = u_exact
    elif BC == 'Robin':
        for i in range(len(r)):
            r[i] = Expression(sym.printing.ccode(r[i]), t=0)
```

```
    def print_error(t, u, timestep):
        """user_action function: print max error at dofs."""
        u_exact.t = t
        u_e = interpolate(u_exact, u.function_space())
        error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
        print('t=%.4f, error: %-10.3E max u: %-10.3f' %
              (t, error, u.vector().array().max()))
        if debug:
            print('u exact:', u_e.vector().array())
        if expect_exact_sol:
            assert error < error_tol, error

    if A_is_const is None:
        A_is_const = BC == 'Dirichlet'
    if lumped_mass:
        assert A_is_const
    # Match dt to N to keep dt/(2*d*dx**q) const,
    # q=1 for theta=0.5 else q=2
    dx = 1./N
    q = 1 if theta == 0.5 else 2
    dt = (0.05/(2*d*0.5**q))*2*d*dx**q
    print('dx=%g, dt=%g (p=1)' % (dx, dt))
    T = 5*dt  # always 5 steps
    if d == 1:
        divisions = (N,)
        L = (1,)
    elif d == 2:
        divisions = (N, N)
        L = (1, 1)
    elif d == 3:
        divisions = (N, N, N)
        L = (1, 1, 1)
    solver(rho, c, p, f, r, s, u_exact, T, L,
           dt, divisions, degree=degree, theta=theta,
           user_action=print_error,
           u0_project=False, BC=BC, A_is_const=A_is_const,
           lumped_mass=lumped_mass,
           avoid_b_assembly=avoid_b_assembly, debug=debug)

def test_efficiency():
    """
    Measure the efficiency of various versions of a Forward
    Euler method: lumped coefficient matrix without any assembly,
    consistent coefficient matrix but no assembly, initial assembly
    of coefficient matrix and assembly of right-hand side,
    and full assembly at each time level.
    Gain: a factor of 6, 4, 2, and 1.
    """
    x, y, z, t = sym.symbols('x[0] x[1] x[2] t')

    # 2D efficiency test: N=180, same gain
    # 3D efficiency test
    u = 1 + x**2 + y + z**2 + 3*t
```

```
    # Lumped 0.14
    verify(u, d=3, degree=1, BC='Dirichlet', N=30, theta=0,
           lumped_mass=True, avoid_b_assembly=True, error_tol=1E-11)
    # No assembly of A and b, just matrix-vector operations 0.22
    verify(u, d=3, degree=1, BC='Dirichlet', N=30, theta=0,
           lumped_mass=False, avoid_b_assembly=True, error_tol=1E-10)
    # Initial assembly of A, assembly of b at each time level 0.42
    verify(u, d=3, degree=1, BC='Dirichlet', N=30, theta=0,
           lumped_mass=False, avoid_b_assembly=False, error_tol=1E-10)
    # Aseembly of A and b the normal way 0.86
    verify(u, d=3, degree=1, BC='Dirichlet', N=30, theta=0,
           lumped_mass=False, avoid_b_assembly=False, A_is_const=False,
           error_tol=1E-10)
```

With the `verify` function we can easily construct a range of unit tests. For example, one test function might look as

```
def test_solver():
    x, y, z, t = sym.symbols('x[0] x[1] x[2] t')

    # 1D
    u = 1 + x**2 + 3*t
    verify(u, d=1, degree=1, BC='Dirichlet', N=20, theta=1)
    verify(u, d=1, degree=2, BC='Dirichlet', N=2, theta=1)
    verify(u, d=1, degree=1, BC='Robin', N=2,  theta=1)
    verify(u, d=1, degree=1, BC='Robin', N=20, theta=1)
    verify(u, d=1, degree=2, BC='Robin', N=2,  theta=1, error_tol=1.5E-13)
    verify(u, d=1, degree=2, BC='Robin', N=2,  theta=0.5)
    # Optimized versions
    verify(u, d=1, degree=1, BC='Dirichlet', N=2, theta=1,
           lumped_mass=True, avoid_b_assembly=True)
    verify(u, d=1, degree=1, BC='Dirichlet', N=2, theta=0.5,
           lumped_mass=True, avoid_b_assembly=True)
    verify(u, d=1, degree=1, BC='Dirichlet', N=2, theta=0,
           lumped_mass=True, avoid_b_assembly=True)

    # 2D
    u = 1 + x - 4*y**2 + 3*t
    verify(u, d=2, degree=1, BC='Dirichlet', N=2, theta=0.5)
    verify(u, d=2, degree=1, BC='Dirichlet', N=2, theta=1)
    verify(u, d=2, degree=1, BC='Dirichlet', N=2, theta=0)
```

Recall that `verify` performs the right `assert` only if the numerical solution is without approximation errors. Robin conditions may lead to exact solutions in 1D, but this is not true in 2D. There, we get a varying r, depending on the manufactured solution, which destroys the exact computation of derivatives. An extension of `verify` is necessary such that we measure convergence rates when `expect_exact_sol` is `False`.

1.1.8 Debugging of FEniCS programs

hpl 3: This section is unfinished.

When the first author implemented the `solver` function at hand, the solution looked nice in visualizations, but the verification tests where the solutions should be reproduced to machine precision, were not fulfilled, although the numerical solutions converged. These observations pointed to bugs in the code, but the author could not spot them from pure reading. How can such a FEniCS code be systematically debugged? The safest way involves the following steps:

1. Reduce the problem to one spatial dimension.
2. Work with P1 elements.
3. Work with the smallest sensible mesh, e.g., two cells.
4. Compute by hand the contribution to the coefficient matrix and right-hand side from each term in the PDEs.
5. Assemble each term in the PDEs individually in FEniCS (easy!) for comparison with hand calculations. (Be aware of the `vertex_to_dof` mapping in FEniCS!)
6. Write out all the Dirichlet conditions and check that they are correct.
7. Finally assert that the linear system computed by hand and by FEniCS are identical.

This procedure requires, of course, that one masters the basic algorithms in the finite element method and can perform these by hand or by a separate program. The details in the present PDE application are documented next.

We start with reducing the problem to 1D. There are four types of terms in our PDE: the mass matrix term $\int \varrho c \frac{1}{\Delta t} u^{n+1} v$, the stiffness matrix term $\int p \nabla u \cdot \nabla v$, the source term $\int f v$, and the Robin condition term $\int_\Gamma r(u-s)v$. We must compute the element matrix or vector for each of these terms and assemble the corresponding matrix or vector. Alternatively, we may compute the matrix or vector directly, without considering element contributions. We prefer the former approach here.

Numbering of the unknowns. Before diving into the calculations, we must know what type of mesh our FEniCS code works with. The potential issue is how the unknowns in the linear system (i.e., the degrees of freedom or dofs) are numbered, see Section 6.1.4 in [2]. While hand calculations typically prefer a numbering from left to right, FEniCS may employ more sophisticated numberings. Each cell has two vertices with numbers. For P1 elements, where the degrees of freedom coincide with the function values at the vertices, we need to figure out what the dof numbering is:

```
>>> from fenics import *
>>> mesh = UnitIntervalMesh(2)
>>> for i, p in enumerate(mesh.coordinates()):  # vertex numbering
...   print(i, p)
```

```
...
0 [ 0.]
1 [ 0.5]
2 [ 1.]
>>> V = FunctionSpace(mesh, 'P', 1)
>>> print vertex_to_dof_map(V)
[2 1 0]
```

We see that vertex number 0 corresponds to dof 2, vertex 1 to dof 1, and vertex 2 to dof 0. This is very important to remember when doing the hand calculations!

Hand calculations. The mass matrix term $\int \varrho c \frac{1}{\Delta t} u^{n+1} v$ leads to an integral over each element involving the finite element basis functions: $\int \varrho c \frac{1}{\Delta t} \phi_i \phi_j \,\mathrm{d}x$. The corresponding element matrix becomes (constant ϱc):

$$\varrho c \frac{h}{6\Delta t} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

The parameter h is the length of an element. Our computational case consists of two elements only, so $h = \frac{1}{2}$ and the global mass matrix reads

$$\varrho c \frac{h}{6\Delta t} \begin{pmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

The corresponding computation in FEniCS is

```
F_M = rho*c*(u-v)/dt*v*dx
if debug:
    M = assemble(lhs(F_M))  # assemble rho*c*u/dt*v*dx
    print('Mass matrix:\n', M.array())
```

The "backward" dof numbering of unknowns used in FEniCS in this case does not influence the assembly by hand of the two element matrices.

The stiffness matrix term $\int p \nabla u \cdot \nabla v$ leads to an element-wise integral $\int \phi_i' \phi_j' \,\mathrm{d}x$ when $p = 1$ and the associated element matrix

$$\frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

The assembled, global matrix becomes

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$$

Also here, there is no impact of the dof numbering in FEniCS.

The Robin condition in 1D reduces to

$$\int_\Gamma r(u-s)v \,\mathrm{d}s = [r(u-s)v]_0^1,$$

which gives a contribution $[ruv]_0^1$ to the coefficient matrix and a contribution $[rsv]_0^1$ to the right-hand side vector. We have $[ruv]_0^1 = r(1)u(1)v(1) - r(0)u(0)v(0)$. The first term gives a contribution to the dof that corresponds to $x = 1$ only, since $\phi_i(1)\phi_j(1) \neq 0$ iff i and j is the dof at $x = 1$. We typically get the global matrix

$$r(1,t)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

when dof 0 corresponds to the point (vertex) $x = 1$. Now the FEniCS numbering of dofs becomes important! The term $r(0)u(0)v(0)$ gives a similar contribution

$$r(0,t)\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

to the global matrix. The corresponding contributions to the right-hand side vector are

$$r(1,t)s\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad r(0,t)s\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Our final term to be computed is $\int fv$, which in the case $f = 1$ gives rise to element integrals $\int \phi_i \, \mathrm{d}x$ and the element vector $(h/2)(1,1)$. The assembled, global vector for two elements reads $h(\frac{1}{2}, 1, \frac{1}{2})$.

Comparing hand and FEniCS calculations. Now we are ready to see if the hand calculations correspond with those in FEniCS. With the variational form split into different pieces, it is easy to assemble each piece individually and print the corresponding matrix or vector:

```
def D(u):
    return p*dot(grad(u), grad(v))*dx

def B(u, i):
    return r[i]*(u-s)*v*ds(i)

U = theta*u + (1-theta)*u_n
F_M = rho*c*(u-u_n)/dt*v*dx
F_K = D(U)
F_f = f*v*dx
F = F_M + F_K - F_f
if BC == 'Robin':
    # Add cooling condition integrals from each side
    F_R = sum(B(U, i) for i in range(2*d))
    F += F_R

if debug:
```

```
    print('M:\n', assemble(lhs(F_M)).array())
    print('K:\n', assemble(lhs(F_K)).array())
    print('R:\n', assemble(lhs(F_R)).array())
    print('A:\n', assemble(lhs(F)).array())
    print('rhs M:', assemble(rhs(F_M)).array())
    print('rhs f:', assemble(rhs(F_f)).array())
    print('rhs R:', assemble(rhs(F_R)).array())
    print('b:', assemble(rhs(F)).array())
a, L = lhs(F), rhs(F)
```

Start with a case with Dirichlet conditions and then proceed with testing Robin conditions.

1.1.9 Avoiding all assembly

The purpose of this section is to present a technique for speeding up FEniCS simulators for time-dependent problems where it is possible to perform all assembly operations prior to the time loop. There are two costly operations in the time loop: assembly of the right-hand side b and solution of the linear system via the `solve` call. The assembly process involves work proportional to the number of degrees of freedom N, while the solve operation has a work estimate of $\mathcal{O}(N^\alpha)$, for some $\alpha \geq 1$. Typically, $\alpha \in [1,2]$. As $N \to \infty$, the solve operation will dominate for $\alpha > 1$, but for the values of N typically used on smaller computers, the assembly step may still represent a considerable part of the total work at each time level. Avoiding repeated assembly can therefore contribute to a significant speed-up of a finite element code in time-dependent problems.

Deriving recursive linear systems. To see how repeated assembly can be avoided, we look at the "right-hand side part" of the variational form (i.e., the linear form $L(v)$) when, for simplicity, $\theta = 1$:

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

This expression varies in general with time through u^n, f^{n+1}, and possibly also with Δt if the time step is adjusted during the simulation. The technique for avoiding repeated assembly consists in expanding the finite element functions in sums over the basis functions ϕ_i to identify matrix-vector products that build up the complete system. We have $u^n = \sum_{j=1}^N U_j^n \phi_j$, and we can expand f^n as $f^n = \sum_{j=1}^N F_j^n \phi_j$. Inserting these expressions in $L(v)$ and using $v = \phi_i$ result in

hpl 4: Why $hat\phi_i$? No need for Petrov-Galerkin here... Cannot remember why the hat. Remove it.

$$\int_\Omega \left(\frac{1}{\Delta t}u^n + f^{n+1}\right) v \,\mathrm{d}x = \int_\Omega \left(\frac{1}{\Delta t}\sum_{j=1}^N U_j^n \phi_j + \sum_{j=1}^N F_j^{n+1}\phi_j\right)\phi_i \,\mathrm{d}x,$$
$$= \sum_{j=1}^N \frac{1}{\Delta t}\left(\int_\Omega \phi_i\phi_j \,\mathrm{d}x\right) U_j^n + \sum_{j=1}^N \left(\int_\Omega \phi_i\phi_j \,\mathrm{d}x\right) F_j^{n+1}.$$

Introducing $M_{ij} = \int_\Omega \phi_i\phi_j \,\mathrm{d}x$, we see that the last expression can be written

$$\sum_{j=1}^N \frac{1}{\Delta t} M_{ij} U_j^n + \sum_{j=1}^N M_{ij} F_j^{n+1},$$

which is nothing but two matrix-vector products,

$$\frac{1}{\Delta t} MU^n + MF^{n+1},$$

if M is the matrix with entries M_{ij},

$$U^n = (U_1^n, \ldots, U_N^n)^T,$$

and

$$F^{n+1} = (F_1^{n+1}, \ldots, F_N^{n+1})^T.$$

We have immediate access to U^n in the program since that is the vector in the `u_n` function. The F^{n+1} vector can easily be computed by interpolating the prescribed f function (at each time level if f varies with time). Given M, U^n, and F^{n+1}, the right-hand side b can be calculated as

$$b = \frac{1}{\Delta t} MU^n + MF^n.$$

That is, no assembly is necessary to compute b!

Generalization to the full model. It now remains to extend the results to the full θ rule and to the boundary terms arising from the Robin conditions. Looking at (1.4), inserting

$$U = \theta \sum_j \phi_j U_j + (1-\theta)\sum_j \phi_j U_j^n,$$

and utilizing that $p\nabla U \cdot \nabla v$ and $r(U-s)v$ are linear in U, we get a right-hand side contribution

$$b = \frac{1}{\Delta t} MU^n + \theta MF^n - (1-\theta)KU^n - (1-theta)RU^n - g, \tag{1.5}$$

where R is the matrix arising from the Robin condition:

$$R_{i,j} = \int_\Gamma r\phi_i\phi_j \,\mathrm{d}s,$$

and g is the associated vector,

$$g_i = \int_\Gamma rs\phi_i \,\mathrm{d}s\,.$$

Splitting the coefficient matrix. If we decide to use a varying time step Δt, the A matrix will vary with time, but it has a special structure so that it can easily and cheaply be computed at each time level. To see this, we insert $v = \phi_i$ and $u^n = \sum_{j=1}^N U_j^n\phi_j$ in the bilinear expression for the simplified case $\theta = 1$ and no Robin conditions to get

$$\sum_{j=1}^N \left(\int_\Omega \frac{1}{\Delta t}\phi_i\phi_j \,\mathrm{d}x\right) U_j^n + \sum_{j=1}^N \left(\int_\Omega \nabla\phi_i\cdot\nabla\phi_j \,\mathrm{d}x\right) U_j^n,$$

which can be written as a sum of matrix-vector products,

$$\frac{1}{\Delta t}MU^n + KU^n = (\frac{1}{\Delta t}M + \Delta t K)U^n,$$

if we identify the matrix M with entries M_{ij} as above and the matrix K with entries

$$K_{ij} = \int_\Omega \nabla\phi_i\cdot\nabla\phi_j \,\mathrm{d}x\,. \tag{1.6}$$

The matrix M is often called the "mass matrix" while "stiffness matrix" is a common nickname for K. The associated bilinear forms for these matrices, as we need them for the assembly process in a FEniCS program, become

Mathematics	FEniCS Code
$a_K(u,v) = \int_\Omega \nabla u\cdot\nabla v \,\mathrm{d}x$	`a_K = dot(nabla(u), nabla(v))*dx`
$a_M(u,v) = \int_\Omega uv \,\mathrm{d}x$	`a_M = u*v*dx`

The linear system at each time level, written as $AU^n = b$, can now be computed by first computing M and K, and then forming $A = \Delta t^{-1}M + K$ at $t = 0$, while b is computed as $b = \Delta t^{-1}MU^{n-1} + MF^n$ at each time level.

hpl 5: Using u^n and u^{n-1}, should be $n+1$ and n.

Generalization to full model. The coefficient matrix associated with the complete variational form (1.4) leads to somewhat more complicated formulas. Just for simplicity, we drop this optimization when we have Robin conditions only. Ignoring the integral over the boundary, we get that

$$A = \varrho c\frac{1}{\Delta t}M + \theta K,$$

and

$$b = \varrho c \frac{1}{\Delta t} M u^n + M F^m - (1-\theta) K u^n,$$

where $F^{n+\theta}$ is the vector of interpolated f values at time $t_{n+\theta} = f(\theta t_{n+1} + (1-\theta)t_n)$.

hpl 6: Not finished.

FEniCS implementation. It is tempting to construct A as

```
A = rho*c*(1./dt)*M + theta*K
```

but that statement invokes a problem: matrix arithmetics works with scalars only, while `rho`, and `c` are `Constant` objects. We therefore need to convert them to plain real numbers:

```
A = float(rho)*float(c)*(1./dt)*M + theta*K
```

Assume that we have already made the mass and stiffness matrices prior to the time loop,

```
if avoid_b_assemble:
    M = assemble(u*v*dx)
    K = assemble(D(u))
```

we can in the time loop write

```
if avoid_b_assemble:
    assert BC == 'Dirichlet'  # restrict for simplicity
    f_m = interpolate(f, V)
    F_m = f_m.vector()
    A = float(rho)*float(c)*(1./dt)*M + theta*K
    b = float(rho)*float(c)*(1./dt*M*u_n.vector() + \
        M*F_m - (1-theta)*K*u_n.vector()
else:
    # Assume A is assembled initially as A = assemble(lhs(F))
    b = assemble(L, tensor=b)
```

That is, no assembly at all is needed inside the time loop, just matrix-vector operations. We hope that this can give a performance boost, but experiments to be reported later show that the gain is about a factor of 4 compared to full assembly of A and b the usual way.

Finished here. [[[

The `user_action` function `assert_error` asserts equality of the exact and numerical solution at every time level:

```
def assert_error(t, u, timestep):
    u_e = interpolate(u_b, u.function_space())
    error= np.abs(u_e.vector().array() -
                  u.vector().array()).max()
    tol = 2E-12
    assert error < tol, 'error: %g' % error
```

One can also use the user action callback function to visualize the solution:

```
def assert_error(t, u, timestep):
    global p
    if t == 0:
        p = plot(u, title='u',
             # Fix the color scale
                 range_min=float(u_range[0]),  # must be float
                 range_max=float(u_range[1]))  # must be float
    else:
        p.plot(u)
    print('t=%g' % t)
    time.sleep(0.5)
```

It is key to fix the color scale to get a meaningful animation.

A complete function calling up `solver_minimize_assembly` for animating the solution in two test problems is found in the function `application_animate` in `ft12_heat_func.py`.

1.1.10 Lumped mass matrix

What is the problem with the mass matrix? Comparing a P1 finite element method with a standard finite difference method, on a uniformly partitioned hypercube mesh, reveals that the stiffness matrix K, arising from $\int p\nabla u\cdot\nabla v\,\mathrm{d}x$, is the same for both methods, while the mass matrix M, arising from a term like $\int uv\,\mathrm{d}x$, is very different in the two methods. The mass matrix is diagonal in the finite difference method, thus making $\theta=0$ a truly explicit scheme with no need for solving a system of linear algebraic equations in each time step. It would be very convenient to have this mass matrix also in finite element solvers to speed up computations. Moreover, analysis of the damping properties of the finite element and finite difference methods in diffusion problems $\frac{\partial u}{\partial t}=\alpha\frac{\partial^2 u}{\partial x^2}$ shows that the finite difference method, with its diagonal mass matrix, is more accurate than the finite element method. For $\theta=0$ it also has less strict stability properties.

To be more precise with the mentioned analysis, a Fourier component $u(x,t)=A(t)\sin(kx)$ of the solution is damped by a factor A from one time step to the next ($A=e^{-\alpha k^2 t}$). It turns out that in a P1 finite element method or a finite difference method with second-order, centered spatial differences, A depends on $p=kh/2$ and $F=\alpha\Delta t/h^2$, where h is the cell length. The dimensionless number F is known as the mesh Fourier number. The Forward Euler scheme, $\theta=0$, is unstable unless $F\leq 1/6$ in the finite element method and $F\leq 1/2$ in the finite difference method. Figure 1.1 compares the graphs of A for the two methods an values of F that give stable solutions. We clearly see that the finite element method gives rise to $A<0$ for short waves, which is manifested as "flipping noise" in animations where the solution oscillates

from time step to time step. In very smooth solutions, the short waves have too small amplitudes for this effect to be visible, but in problems with discontinuities, it is easy to spot that reducing F increases the noise.

Figures 1.2 and 1.3 compares A for a wider range of F values for the Backward Euler and Crank-Nicolson schemes, respectively. Also here we clearly see that the finite element method leads to inferior damping properties compared to the finite difference method. "Flipping noise" for high frequencies (due to $A < 0$) is a well-known flaw in the Crank-Nicolson scheme, but the negative effect is more pronounced in the finite element method. However, it is easy to come up with a remedy for the finite element method: if we make the mass matrix diagonal, the method will be equivalent to the finite difference method on a uniformly partitioned mesh and hence equally accurate.

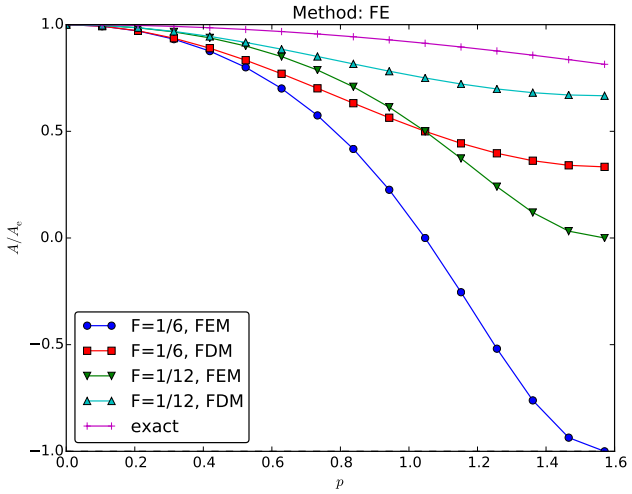

Fig. 1.1 Comparison of damping factors for the Forward Euler scheme.

Lumping the mass matrix. A widely used method for making the mass matrix diagonal, or *lump* it as the phrase often goes[1], is to sum all elements in a row, set the sum on the diagonal, and put zeros in all the other columns. This is called the row-sum technique and applies well to P1 elements, but it does not work well with P2 and higher-order elements. For these elements, one should instead use a quadrature that only samples the integrands at the points where we have the function value degrees of freedom. This is not yet

[1]The term arose in the days where the primary application was wave motion (in elastic or other media) and the diagonalization consisted in concentrating uniformly distributed mass as lumps at the nodes only.

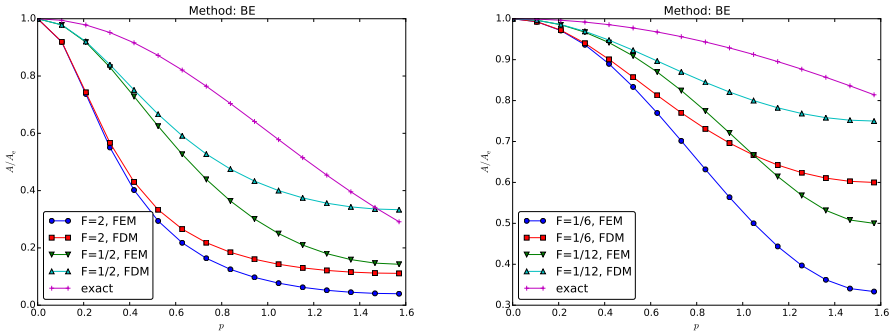

Fig. 1.2 Comparison of damping factors for the Backward Euler scheme.

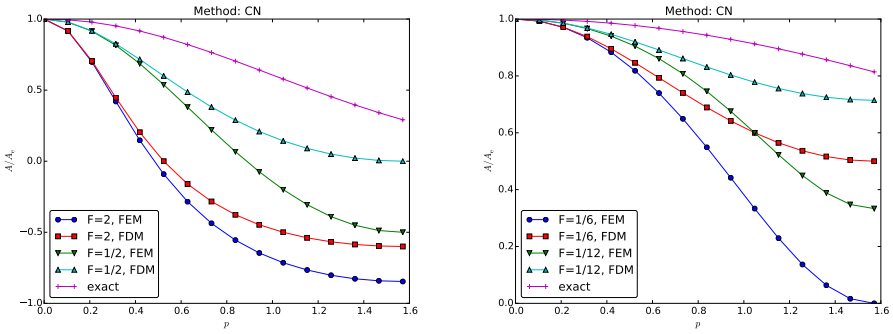

Fig. 1.3 Comparison of damping factors for the Crank-Nicolson scheme.

possible in FEniCS **hpl 7**: Anders, right? so lumping remains restricted to P1 elements as explained here.

Implementation in FEniCS. The row sum can be computed by multiplying the matrix by a vector with 1 for all elements,

```
unity = Function(V)
unity.vector()[:] = 1.

[bc.apply(M, b) for bc in bcs]
ML = M*unity.vector()  # lump M
```

It is very important that we carry out the multiplication with `unity` (i.e., the row sum) *after* Dirichlet boundary conditions are inserted in the matrix `M`! Otherwise, the boundary conditions will not be incorporated in the vector `ML`.

We introduce a boolean `lumped_mass` in the `solver` function for indicating computations with a lumped mass matrix. Prior to the time loop we need to make sure we have the `unity` field available as well as `M` and `K` for the mass matrix and the stiffness matrix:

```
if lumped_mass:
```

```
    M = assemble(u*v*dx)
    K = assemble(D(u))
```

We need to distinguish between two cases: $\theta = 0$ and $\theta > 0$. In the former case, the entire coefficient matrix is lumped and we can avoid calling `solve`, while in the latter case, parts of the coefficient matrix involves a lumped mass matrix while other parts employ standard, assembled matrices. For $\theta = 0$ we create the coefficient matrix as the matrix `rho*c*M` or

```
if lumped_mass and theta == 0:
    A = assemble(lhs(F_M)) # make consistent mass, lump later
else:
    A = assemble(lhs(F))
```

For $\theta > 0$, the implementation is a bit tricky. We would like to write

```
ML = M*unity.vector()  # lump
A1 = float(rho)*float(c)*(1./dt)*ML
A = A1 + theta*K
```

but `A1 + theta*K` does not work because it is a vector plus a matrix. The trick is to add `A1` to the diagonal of `theta*K`:

```
A2 = theta*K
diag = unity.vector().copy()  # allocate vector
A2.get_diagonal(diag)
A2.set_diagonal(diag + A1)
A = A2
```

Now, `A` is a matrix, but with a lumped mass matrix contribution.

The right-hand side $b = \varrho c \Delta t^{-1} M u^n + M F^m - (1-\theta)K$ is for a lumped mass matrix and $\theta > 0$ computed as

```
b = float(rho)*float(c)*1./dt*ML*u_n.vector() + \
    ML*F_m - (1-theta)*K*u_n.vector()
```

We have decided to put all the mentioned methods together in the `solver` function, and together with debug output, this makes a quite lengthy function in file `ft12_heat_func.py`. There are four methods for different kinds of assembly and three boolean variables to control them:

- `lumped_mass`: `True` if the mass matrix is lumped
- `avoid_b_assembly`: `True` if the right-hand side is to be computed from matrix-vector products and no assembly
- `A_is_const`: `True` if the coefficient matrix is constant (assumption when using lumped mass).

Four methods are available for assembling the coefficient matrix A and right-hand side b:

1. Full assembly of A and b at each time level: `lumped_mass=False`, `avoid_b_assembly=False`, `A_is_const=False`

2. Initial assembly of A, full assembly of b at each time level: `lumped_mass=False`, `avoid_b_assembly=False`, `A_is_const=True`
3. No assembly of A and b at each time level, just construction through matrix-vector products: `lumped_mass=False`, `avoid_b_assembly=True`, `A_is_const=True`
4. No assembly of A and b at each time level, just construction through matrix-vector products, but with lumped mass matrices: `lumped_mass=True`, `avoid_b_assembly=True`, `A_is_const=True`

The function `test_efficiency` measures the impact of the four optimization technique using the verification problem from Section 1.1.7. The results are independent of the number of space dimensions and the number of unknowns. We find typically for $\theta = 0$ that method 2-4, compared to method 1, speeds up the code by a factor of 2, 4, and 6, respectively. The factor 2 is easy to explain: if the work of creating A and b by assembly is about the same, we gain a factor of two by omitting one of them at every time step. One would perhaps expect that the factors 4 and 6 would be better, but in these tests, we have used a sparse matrix representation of A, and matrix–vector arithmetics with sparse matrices are demanding to implement efficiently because of the many cache misses, a problem shared with the assembly algorithm. It may well happen that both algorithms suffer from spending most of the time grabbing data from memory and not on computing (where matrix–vector arithmetics should involve far fewer operations than the assembly algorithm). The trick with manipulating the diagonal of `A2` is also more costly than tailored support for lumped mass matrices in FEniCS. Anyway, a factor 6 may be important in many applications, and the increased accuracy for less smooth solutions of the diffusion equations, as provided by a lumped mass matrix, may often be more important.

Application: diffusion of a spike. Just to demonstrate how to use the `solver` function in an application, we consider pure diffusion of a spike

$$u_0(x,y) = sin^8(\pi x)\sin^8(\pi y),$$

with boundary conditions $u = 0$. We solve the homogeneous, constant-coefficient, scaled diffusion equation, so $p = c = \varrho = 1$, $f = 0$. Moreover, r is a list of `Constant(0)` objects for the Dirichlet conditions (length 4, one `Constant(0)` for each side of the 2D domain). In this case, s is a just a dummy variable but needs to be of type `Expression` or `Constant`, so we choose `Constant(0)`. The domain $[0,1]\times[0,1]$ is divided into a 60×60 partitioning with P1 elements.

We want to animate the solution using the FEniCS built-in `plot` command. The necessary actions must be done in the `user_action` function, here called `animate`.

The application code reads

```
def animate_sine_spike():
```

```
import numpy as np, time

# Diffusion of a sin^8 spike, scaled homogeneous PDE
u0 = Expression('pow(sin(pi*x[0])*sin(pi*x[1]), 8)')
c = rho = p = Constant(1)
f = Constant(0)
r = [Constant(0) for i in range(2*2)] # Dirichlet conditions u=0
s = Constant(0)                       # dummy, not used
dt = 0.0005
T = 20*dt
L = [1, 1]
divisions = (60, 60)
u_range = [0, 1]

vtkfile = File('diffusion.pvd')

def animate(t, u, timestep):
    global plt
    if t == 0:
        plt = plot(u, title='u',
                   range_min=float(u_range[0]),  # must be float
                   range_max=float(u_range[1]))  # must be float
    else:
        plt.plot(u)
    print('t=%g' % t)
    time.sleep(0.5)           # pause between frames
    vtkfile << (u, float(t))  # store time-dep Function

solver(
    rho, c, p, f, r, s, u0, T, L,
    dt, divisions, degree=1, theta=1,
    user_action=animate,
    u0_project=False,
    lumped_mass=False,
    BC='Dirichlet',
    A_is_const=True,
    avoid_b_assembly=True)
```

A little trick is needed in the `animate` function. To fix the color scale throughout the animation, we must initially provide the range of u values for the scale, while later we must update the plot using the object returned from initial `plot` call. This object must survive between successive calls to `animate` so it cannot be a local variable. The solution is to let it be a global variable `plt` (or one could implement `animate` via a class that has `plt` as an attribute and the callback function as method, usually `__call__`).

1.2 A welding example with post processing and animation

The focus so far in this tutorial has been on producing the solution of PDE problems. **hpl 8**: This is book 2, it depends on how things end up in the previous chapter. For scientific investigations, the primary work is often with post processing results: computing quantities derived from the solution and inspecting these with visualization or data analysis tools. This is the focus of the present section. To ease the programming, we shall make use of a convenient tool, `cbcpost`, for post processing, saving data to file(s), and animating solutions. We recommend to use `cbcpost` in all time-dependent FEniCS solvers, but it also has a lot to offer in stationary problems too.

To explain the usage of `cbcpost` for storage and plotting, we address a real physical application: welding of a plate, where a moving heat source gives rise to a moving temperature field.

1.2.1 Post processing data and saving to file

Installation. The `cbcpost` package is not a part of the `fenics` package so you will need to install it. The simplest installation method is to use `pip`. We recommend to install a companion package `fenicstools` as well. Just run

```
sudo pip install git+https://bitbucket.org/simula_cbc/cbcpost.git
sudo pip install git+https://github.com/mikaem/fenicstools.git
```

in a terminal window (skip `sudo` on Windows machines). Alternatively, you can grab the source code and run `setup.py` the usual way Python packages are installed from source:

Terminal

```
Terminal> git clone https://bitbucket.org/simula_cbc/cbcpost.git
Terminal> cd cbcpost
Terminal> python setup.py install
Terminal> cd ..
Terminal> git clone https://github.com/mikaem/fenicstools.git
Terminal> cd fenicstools
Terminal> python setup.py install
```

Basic commands. We must create a *post processor* and then specify what kind of results we want to be stored on file and (optionally) get visualized. Suppose we have a field with logical name `Temperature` that we want to save in XDMF/HDF5 format in files in a fresh subdirectory `Results`:

```
import cbcpost as post
# Create post processor
pp = post.PostProcessor(dict(casedir='Results', clean_casedir=True))
```

```
# Specify storage of a "Temperature" field
pp.add_field(post.SolutionField(
    'Temperature',
    dict(save=True,
         save_as=['hdf5', 'xdmf'],
         plot=True,
         plot_args=dict(range_min=0.0, range_max=1.2))))
```

The `plot=True` automatically launches `fenics.plot` commands of this scalar field during the simulation. The ranges of the color scale must be given (as `float` variables) so that the color scale stays fixed during the animation on the screen.

Inside the time loop, we have to feed a new solution to the post processor to get it saved:

```
pp.update_all({'Temperature': lambda: T}, t, timestep)
```

Here, `T` is the `Function` object that we have solved for, `t` is current time, and `timestep` is the corresponding time step number.

One can specify many fields to be saved (and plotted), but even more important: `cbcpost` can calculate a lot of derived quantities from the solution, such as

- time derivatives and integrals of vector/scalar fields
- extraction of fields over subdomains
- slicing of fields in 3D geometries
- averaging of fields in space or time
- norms and point values of fields as function of time
- user-defined post processing of fields

We refer to the online cbcpost documentation[2] for further information on all the capabilities of this package.

Tip: Use `cbcpost` to visualize time-dependent data

Instead of issuing your own `plot` commands in time-dependent problems, it is safer and more convenient to specify `plot=True` and fix the range of the color scale, when you add fields to the post processor. Multiple fields will be synchronized during the animation.

1.2.2 Heat transfer due to a moving welding source

Let us solve a diffusion problem taken from welding. A moving welding equipment acts as a moving heat source at the top of a thin metal plate. The ques-

[2] `http://cbcpost.readthedocs.org/en/latest/index.html`

tion is how the heat from the equipment spreads out in the material that is being welded. We use the standard heat equation, treat the material as two dimensional, and do not take phase transitions into account. The governing PDE is then

$$\varrho c\frac{\partial u}{\partial t} = \kappa\nabla^2 u + f,$$

where u is temperature, ϱ is the density of the material, c is the heat capacity at constant volume, κ is the heat conduction coefficient, and f models the heat source from the welding equipment. The domain is $\Omega = [0,L]\times[0,L]$. An additional major simplification is that we set $u = U_s$ at the boundary, where U_s is the temperature of the surroundings (a Robin condition, modeling cooling at the boundary would be more accurate, but then we should also consider cooling in the third dimension as well). The initial condition reads $u = U_s$.

A welding source is moving and very localized in space. The localization can be modeled by a peak-shaped Gaussian function. The movement is taken to be a circle with radius R about a point (x_0, y_0). An appropriate f is

$$f(x,y,t) = A\exp\left(-\frac{1}{2\sigma^2}\left(x-(x_0+R\cos\omega t)\right)^2 - \frac{1}{2\sigma^2}\left(y-(y_0+R\sin\omega t)\right)^2\right).$$

The parameter A is the strength of the heat source, and σ is the "standard deviation" (i.e., a measure of the width) of the Gaussian function.

1.2.3 Scaling of the welding problem

There are 10 physical parameters in the problem: L, ϱ, c, κ, A, x_0, y_0, R, ω, σ. Scaling can dramatically reduce the number of parameters and also introduce new parameters that are much easier to assign numerical values when doing numerical experiments. We therefore scale the problem. As length scale, we choose L so the scaled domain becomes the unit square. As time scale and characteristic size of u, we just introduce t_c ad u_c. This means that we introduce scaled variables

$$\bar{x} = \frac{x}{L}, \quad \bar{y} = \frac{y}{L}, \quad \bar{t} = \frac{t}{t_c}, \quad \bar{u} = \frac{u-U_s}{u_c}.$$

The scaled form of f is naturally $\bar{f} = f/A$, since this makes $\bar{f}\in(0,1]$. The arguments in the exponential function in f can also be scaled:

$$\begin{aligned}\bar{f} &= \exp\left(-\frac{1}{2\sigma^2}\left(\bar{x}L-(L\bar{x}_0+L\bar{R}\cos\omega t_c t)\right)^2-\frac{1}{2\sigma^2}\left(L\bar{y}-(L\bar{y}0+L\bar{R}\sin\omega t_c t)\right)^2\right)\\ &= \exp\left(-\frac{1}{2}\frac{L^2}{\sigma^2}\left(x-(\bar{x}_0+\bar{R}\cos\omega t_c\bar{t})\right)^2-\frac{1}{2}\frac{L^2}{\sigma^2}\left(\bar{y}-(\bar{y}0+\bar{R}\sin\omega t_c\bar{t})\right)^2\right)\\ &= \exp\left(-\frac{1}{2}\beta^2\left((x-(\frac{1}{2}+\bar{R}\cos\bar{t})\right)^2-\left(\bar{y}-(\frac{1}{2}+\bar{R}\sin\bar{t}))^2\right)\right),\end{aligned}$$

where β is a dimensionless parameter,

$$\beta = \frac{L}{\sigma},$$

reflecting the ratio of the domain size and the width of the heat source. Moreover, we have restricted the rotation point to be the center point of the domain:

$$(\bar{x}_0, \bar{y}_0) = (\frac{1}{2}, \frac{1}{2}).$$

The time scale in diffusion problems is usually related to the "speed of the diffusion", but in this problem it is more natural to base the time scale on the movement of the heat source, which suggests setting $t_c = 1/\omega$.

Inserting the new scaled variables in the PDE leads to

$$\frac{\partial\bar{u}}{\partial\bar{t}} = \frac{\kappa}{\omega\varrho cL^2}\bar{\nabla}^2\bar{u}+\frac{A}{\omega u_c\varrho c}\bar{f}(\bar{x},\bar{y},\bar{t}).$$

The first coefficient is a dimensionless number,

$$\gamma = \frac{\kappa}{\omega\varrho cL^2},$$

while the second coefficient can be used to determine u_c by demanding the source term to balance the time derivative term,

$$u_c = \frac{A}{\omega\varrho c}.$$

Our aim is to have $\bar{u}\in[0,1]$, but this u_c does not capture the precise magnitude of u. However, we believe that the characteristic size of u is

$$u_c = \delta^{-1}\frac{A}{\omega\varrho c},$$

for a scaling factor δ. Using this u_c gives the PDE

$$\frac{\partial\bar{u}}{\partial\bar{t}} = \gamma\bar{\nabla}^2\bar{u}+\delta\bar{f}(\bar{x},\bar{y},\bar{t}),$$

with two dimensionless variables, but δ is quite easily tuned from experiments to give $\bar{u}$ a typically size of unity.

Looking at γ, we see that it can be written

$$\gamma = \frac{1/\omega}{\varrho c L^2/\kappa},$$

which is the ratio of the time scale for the heat source and the time scale for diffusion. Multiplying by R/R gives another interpretation: γ is the ratio of the speed of diffusion and the speed of the heat source.

The benefits of scaling

The physics of our problem depends now on β, $\bar{R}$, and γ, just three ratios of physical effects instead of 10 independent parameters. Setting $\bar{R} = 0.2$ is an appropriate choice. For a quite localized heat source in space, $\beta = 10$ is a suitable value. Then we are actually left with only one interesting parameter to adjust: γ. It is so much easier to assign this parameter a value (speed of diffusion versus speed of heat source) than to set ϱ, c, and κ for some chosen material, and then determine relevant values for A, L, etc. There are no approximations in the scaling procedure; it just dramatically simplifies numerical simulations. The book [3] gives a comprehensive treatment of scaling.

1.2.4 A function-based solver

We can use the `solver_minimize_assembly` function to solve the welding problem. The application code just declares the problem-dependent parameters and calls the solver function:

```
def application_welding(gamma=1, delta=1, beta=10, num_rotations=2):
    """Circular moving heat source for simulating welding."""
    from math import pi, sin, cos
    u_b = Constant(0)
    I = Constant(0)
    R = 0.2
    f = Expression(
        'delta*exp(-0.5*pow(beta,2)*(pow(x[0]-(0.5+R*cos(t)),2) + '
                                    'pow(x[1]-(0.5+R*sin(t)),2)))',
        delta=delta, beta=beta, R=R, t=0)
    # Simulate to rotations with the equipment
    omega = 1.0      # Scaled angular velocity
    P = 2*pi/omega   # One period of rotation
    T = 2*P          # Total simulation time
    dt = P/40        # 40 steps per rotation
    Nx = Ny = 60
    solver_minimize_assembly(
```

```
        gamma, f, u_b, I, dt, T, (Nx, Ny), (1, 1), degree=1,
        user_action=ProcessResults(), I_project=False)
```

The remaining task is to write the user action callback function to process the solution at teach time step. We want to make use of `cbcpost` for storage and plotting. Since we need the post processor variable, called `pp` in Section 1.2.1, to survive between calls to the user action function, we find it most convenient to implement this function in terms of a class with `pp` as attribute and `__call__` as the user action function. We want to make comparisons between the heat source and the temperature response, so we register both fields for storage and plotting:

```
import cbcpost as post
class ProcessResults(object):
    def __init__(self):
        """Define fields to be stored/plotted."""
        self.pp = post.PostProcessor(
            dict(casedir='Results', clean_casedir=True))
        self.pp.add_field(
            post.SolutionField(
                'Temperature',
                dict(save=True,
                     save_as=['hdf5', 'xdmf'],  # format
                     plot=True,
                     plot_args=
                     dict(range_min=0.0, range_max=1.1)
                     )))
        self.pp.add_field(
            post.SolutionField(
                "Heat_source",
                dict(save=True,
                     save_as=["hdf5", "xdmf"],  # format
                     plot=True,
                     plot_args=
                     dict(range_min=0.0, range_max=float(delta))
                     )))
        # Save separately to VTK files as well
        self.vtkfile_T = File('temperature.pvd')
        self.vtkfile_f = File('source.pvd')
    def __call__(self, t, T, timestep):
        """Store T and f to file (cbcpost and VTK)."""
        T.rename('T', 'solution')
        f_Function = interpolate(f, T.function_space())
        f_Function.rename('f', 'welding equipment')
        self.pp.update_all(
            {'Temperature': lambda: T,
             'Heat_source': lambda: f_Function},
            t, timestep)
        self.vtkfile_T << (T, float(t))
        self.vtkfile_f << (f_Function, float(t))
```

We took the opportunity to also store the `u` and `f` functions to VTK files, although this is really not necessary since ParaView or VisIt can read XDMF files.

Note that the use of `cbcpost` is usually very dependent on the problem at hand, so it does not make sense to include `cbcpost` code in a general PDE solver, only in problem-specific code such as the user action function.

Getting an animation on the screen with the built-in plotting tool is a matter of running the welding example:

```
>>> from heat_func import application_welding as a
>>> a(gamma=10, delta=700)
```

(We introduced the synonym `a` to save some typing.) Or you can run this as a command in the terminal:

Terminal

```
Terminal> python -c '\
from heat_func import application_welding as a;
a(gamma=10, delta=700)'
```

Since we have fixed the color scale of the temperature to have values in $[0, 1.1]$, we must adjust δ appropriately to γ. For example, running $\gamma = 40$ reveals, from the output in the terminal, that the maximum temperature is about 0.25, and consequently we do not see much. For any given γ, run the problem with $\delta = 1$ (and say `num_rotations=0.2` to make a quick simulation), and rerun with δ as one over the maximum temperature. Here we get an approximate $\delta = 66.7\gamma$ for $\gamma \leq 0.1$. Try running $\gamma = 0.01$ and $\delta = 1$ to observe some more significant heat transfer away from the welding equipment. With $\gamma = 0.001$ there is significant heat build-up, but for so small γ we should re-scale the problem and use the diffusion time scale as time scale.

In ParaView, load `Results/Temperature/Temperature.xdmf` as file, click **Apply**, then the play button for animation. If the animation is not correct, repeat the procedure. Thereafter, split the layout in two, choose **3D View**, load `Results/Heat source/Heat_source.xmdf`, click **Apply**, and run the animation. The two plots are synchronized in time.

Movie 1: Welding example with $\gamma = 1$. `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/welding_gamma1.ogg`

Chapter 2
PDE solver design and coding practices

In the very beginning of this tutorial [2] we focused on how to quickly put together solvers for a number of different PDEs. FEniCS makes it simple and straightforward to write the commands needed to set up a variational problem, define domains and boundary conditions. Then we wrapped such flat programs in functions for increased flexibility and easy testing. However, for a real application you will likely want to be able to reuse the code you write for a particular PDE to solve multiple different problems with different domains, boundary conditions and other parameters. In this chapter, we look at how to structure FEniCS Python code to create flexible, reusable, and efficient PDE solvers. The key concept is to use Python classes and develop effective an Application Programming Interface (API) in terms of methods and their arguments.

2.1 Refactoring a Poisson solver in terms of classes

A FEniCS solver for a PDE can be implemented in a general way, but the problem-dependent data, like boundary conditions, must be specified in each case by the user. For example, the implementation in Section 4.3 in [2] requires the user to supply a `boundary_conditions` dictionary with specifications of the boundary condition on each of the four sides of the unit square. If we, e.g., want two Dirichlet conditions at one side, this is not possible without modifying the solver function. What do to with a general mesh is an open question. To avoid changing the code in what is meant to be a general PDE solver for a wide class of problems, we need a different software design.

Such a different design is to introduce a problem class and methods, supplied by the user from case to case, where boundary conditions and other input data are defined. Such a design is used in a lot of more advanced FEn-

iCS application codes, and it is time to exemplify it here. As a counterpart to the solver function, we introduce a solver class, but all the arguments for various input data are instead method calls to an instance of a *problem class*. This puts a somewhat greater burden on the programmer, but it allows for more flexibility, and the code for, e.g., boundary conditions can be more tailored to the problem at hand than the code we introduced in the `solver` function in Section 4.3.

2.1.1 The solver class

The solver class will need problem information and for this purpose call up the methods in a problem class. For example, the solver gets the f and p functions in the PDE problem by calling `problem.f_rhs()` and `problem.p_coeff()`. The mesh object and the polynomial degree of the elements are supposed to be returned from `problem.mesh_degree()`. Furthermore, the problem class defines the boundary conditions in the problem as lists of minimal information from which the solver can build proper data structures.

The solver class is a wrapping of the previous `solver_bc` and `flux` functions as methods in a class, but some of the code for handling boundary conditions in `solver_bc` is now delegated to the user in the problem class.

```
from fenics import *
import numpy as np

class PoissonSolver(object):
    def __init__(self, problem, debug=False):
        self.problem = problem
        self.debug = debug

    def solve(self, linear_solver='direct'):
        Dirichlet_cond = self.problem.Dirichlet_conditions()
        if isinstance(Dirichlet_cond, (Expression)):
            # Just one Expression for Dirichlet conditions on
            # the entire boundary
            self.bcs = [DirichletBC(
                V, Dirichlet_cond,
                lambda x, on_boundary: on_boundary)]
        else:
            # Boundary SubDomain markers
            self.bcs = [
                DirichletBC(V, value, boundaries, index)
                for value, boundaries, index
                in Dirichlet_cond]

        if self.debug:
            # Print the Dirichlet conditions
            print('No of Dirichlet conditions:', len(self.bcs))
            coor = self.mesh.coordinates()
```

```
            d2v = dof_to_vertex_map(V)
            for bc in self.bcs:
                bc_dict = bc.get_boundary_values()
                for dof in bc_dict:
                    print('dof %2d: u=%g' % (dof, bc_dict[dof]))
                    if V.ufl_element().degree() == 1:
                        print('   at point %s' %
                              (str(tuple(coor[d2v[dof]].tolist()))))
        # Compute solution
        self.u = Function(self.V)

        if linear_solver == 'Krylov':
            solver_parameters = {'linear_solver': 'gmres',
                                 'preconditioner': 'ilu'}
        else:
            solver_parameters = {'linear_solver': 'lu'}

        self.define_variational_problem()
        solve(self.a == self.L, self.u, self.bcs,
              solver_parameters=solver_parameters)
        return self.u

    def define_variational_problem(self):
        self.mesh, degree = self.problem.mesh_degree()
        self.V = V = FunctionSpace(self.mesh, 'P', degree)

        u = TrialFunction(V)
        v = TestFunction(V)
        p = self.problem.p_coeff()
        f = self.problem.f_rhs()
        F = dot(p*grad(u), grad(v))*dx
        F -= f*v*dx
        F -= sum([g*v*ds_
                  for g, ds_ in self.problem.Neumann_conditions()])
        F += sum([r*(u-s)*ds_
                  for r, s, ds_ in self.problem.Robin_conditions()])
        self.a, self.L = lhs(F), rhs(F)

        if self.debug and V.dim() < 50:
            A = assemble(self.a)
            print('A:\n', A.array())
            b = assemble(self.L)
            print('b:\n', b.array())

    def flux(self):
        """Compute and return flux -p*grad(u)."""
        mesh = self.u.function_space().mesh()
        degree = self.u.ufl_element().degree()
        V_g = VectorFunctionSpace(mesh, 'P', degree)
        p = self.problem.p_coeff()
        self.flux_u = project(-p*grad(self.u), V_g)
        self.flux_u.rename('flux(u)', 'continuous flux field')
        return self.flux_u
```

```
class PoissonProblem(object):
    """Abstract base class for problems."""
    def solve(self, linear_solver='direct',
              abs_tol=1E-6, rel_tol=1E-5, max_iter=1000):
        self.solver = PoissonSolver(self)
        prm = parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        return self.solver.solve(linear_solver)

    def solution(self):
        return self.solver.u

    def mesh_degree(self):
        """Return mesh, degree."""
        raise NotImplementedError('Must implement mesh!')

    def p_coeff(self):
        return Constant(1.0)

    def f_rhs(self):
        return Constant(0.0)

    def Dirichlet_conditions(self):
        """Return list of (value,boundary_parts,index) triplets,
        or an Expression (if Dirichlet values only)."""
        return []

    def Neumann_conditions(self):
        """Return list of (g,ds(n)) pairs."""
        return []

    def Robin_conditions(self):
        """Return list of (r,u,ds(n)) triplets."""
        return []
```

Note that this is a general Poisson problem solver that works in any number of space dimensions and with any mesh and composition of boundary conditions!

Tip: Be careful with the `mesh` variable!

In classes, one often stores the mesh in `self.mesh`. When you need the mesh, it is easy to write just `mesh`, but this gives rise to peculiar error messages, since `mesh` is a Python module imported by `from fenics import *` and already available as a name in your file. When encountering strange error messages in statements containing a variable `mesh`, make sure you use `self.mesh`.

2.1.2 A problem class

Let us start with a relatively simple problem class for our favorite test problem where we manufacture a solution $u_{\mathrm{D}} = 1 + x^2 + 2y^2$ and solve $-\nabla^2 u = f$ with $f = 6$ and $u = u_{\mathrm{D}}$ at the entire boundary.

```
class TestProblemExact(PoissonProblem):
    def __init__(self, Nx, Ny):
        """Initialize mesh, boundary parts, and p."""
        self.mesh = UnitSquareMesh(Nx, Ny)
        self.u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

    def mesh_degree(self):
        return self.mesh, 1

    def f_rhs(self):
        return Constant(-6.0)

    def Dirichlet_conditions(self):
        return self.u_D
```

We can then make a simple unit test for the problem and solver class:

```
def test_PoissonSolver():
    """Recover numerial solution to "machine precision"."""
    problem = TestProblemExact(Nx=2, Ny=2)
    problem.solve(linear_solver='direct')
    u = problem.solution()
    u_e = interpolate(problem.u_D, u.function_space())
    max_error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
    tol = 1E-14
    assert max_error < tol, 'max error: %g' % max_error
```

2.1.3 A more complicated problem class

Below is the specific problem class for solving a scaled 2D Poisson problem. We have a two-material domain where a rectangle $[0.3, 0.7] \times [0.3, 0.7]$ is embedded in the unit square and where p has a constant value inside the rectangle and another value outside. On $x = 0$ and $x = 1$ we have homogeneous Neumann conditions, and on $y = 0$ and $y = 1$ we have the Dirichlet conditions $u = 1$ and $u = 0$, respectively.

```
class Problem1(PoissonProblem):
    """
    -div(p*grad(u)=f on the unit square.
    General Dirichlet, Neumann, or Robin condition along each
    side. Can have multiple subdomains with p constant in
    each subdomain.
```

```
    """
    def __init__(self, Nx, Ny):
        """Initialize mesh, boundary parts, and p."""
        self.mesh = UnitSquareMesh(Nx, Ny)

        tol = 1E-14

        class BoundaryX0(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[0]) < tol

        class BoundaryX1(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[0] - 1) < tol

        class BoundaryY0(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[1]) < tol

        class BoundaryY1(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[1] - 1) < tol

        # Mark boundaries
        #self.boundary_parts = FacetFunction('size_t', mesh)
        self.boundary_parts = FacetFunction('uint', self.mesh)
        self.boundary_parts.set_all(9999)
        self.bx0 = BoundaryX0()
        self.bx1 = BoundaryX1()
        self.by0 = BoundaryY0()
        self.by1 = BoundaryY1()
        self.bx0.mark(self.boundary_parts, 0)
        self.bx1.mark(self.boundary_parts, 1)
        self.by0.mark(self.boundary_parts, 2)
        self.by1.mark(self.boundary_parts, 3)
        self.ds =  Measure(
            'ds', domain=self.mesh,
            subdomain_data=self.boundary_parts)

        # The domain is the unit square with an embedded rectangle
        class Rectangle(SubDomain):
            def inside(self, x, on_boundary):
                return 0.3 <= x[0] <= 0.7 and 0.3 <= x[1] <= 0.7

        self.materials = CellFunction('size_t', self.mesh)
        self.materials.set_all(0)  # "the rest"
        subdomain = Rectangle()
        subdomain.mark(self.materials, 1)
        self.V0 = FunctionSpace(self.mesh, 'DG', 0)
        self.p = Function(self.V0)
        help = np.asarray(self.materials.array(), dtype=np.int32)
        p_values = [1, 1E-3]
        self.p.vector()[:] = np.choose(help, p_values)
```

```
    def mesh_degree(self):
        return self.mesh, 2

    def p_coeff(self):
        return self.p

    def f_rhs(self):
        return Constant(0)

    def Dirichlet_conditions(self):
        """Return list of (value,boundary) pairs."""
        return [(1.0, self.boundary_parts, 2),
                (0.0, self.boundary_parts, 3)]

    def Neumann_conditions(self):
        """Return list of g*ds(n) values."""
        return [(0, self.ds(0)), (0, self.ds(1))]
```

A specific problem can be solved by

```
def demo():
    problem = PoissonProblem1(Nx=20, Ny=20)
    problem.solve(linear_solver='direct')
    u = problem.solution()
    u.rename('u', 'potential')  # name 'u' is used in plot
    plot(u)
    flux_u = problem.solver.flux()
    plot(flux_u)
    vtkfile = File('poisson.pvd')
    vtkfile << u
    interactive()

class TestProblemExact(PoissonProblem):
    def __init__(self, Nx, Ny):
        """Initialize mesh, boundary parts, and p."""
        self.mesh = UnitSquareMesh(Nx, Ny)
        self.u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

    def mesh_degree(self):
        return self.mesh, 1

    def f_rhs(self):
        return Constant(-6.0)

    def Dirichlet_conditions(self):
        return self.u_D


def test_PoissonSolver():
    """Recover numerial solution to "machine precision"."""
    problem = TestProblemExact(Nx=2, Ny=2)
    problem.solve(linear_solver='direct')
    u = problem.solution()
    u_e = interpolate(problem.u_D, u.function_space())
```

```
    max_error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
    tol = 1E-14
    assert max_error < tol, 'max error: %g' % max_error

if __name__ == '__main__':
    #demo()
    test_PoissonSolver()
```

The complete code is found in the file `ft10_poisson_class.py`.

Pros and cons of solver/problem classes vs solver function

What are the advantages of class `Solver` and `Problem` over the function implementation in Section 4.3? The primary advantage is that the class version works for any mesh and any composition of boundary conditions, while the solver function is tied to a mesh over the unit square, only one type of boundary condition on a each side, and a piecewise constant p function. The programmer has to supply more code in the class version, but gets greater flexibility. The disadvantage of the class version is that it applies the class concept so one needs experience with Python class programming.

2.2 Refactoring a heat equation solver

The flat program for the diffusion solver in `ft03_heat.py` and `ft11_heat2.py` was refactored in `ft12_heat_func.py` in terms of a `solver` function with the general code for solving the PDE problem, a callback function for processing the solution at each time step, and an application function defining the callback function and calling the solver to solve a specific problem. However, for time-dependent problems a solver function that gets all its input through a set of arguments is less flexible than a solver class, which can demand its input both through arguments and through functions (in subclasses) provided by the user. The following text requires you to be familiar with class programming in Python (tailored learning material is Chapter 7, 9, and Appendix E in [1]).

When we work with a PDE project, we often want to explore a range of similar problems where the PDE model basically stays the same, but coefficients in the PDE, boundary and initial conditions, as well as domains change. This means that some of our code related to solving the PDE is always the same, while some of our code is strongly dependent upon a particular application. To avoid copying code (which is considered evil in computer programming), we need to collect the common code for all problems of this

type in one place and then create an API (application programming interface) to the code that will be different from application to application. To this end, we introduce a *solver class* that applies FEniCS to solve the PDE. It requires access to a *problem class* where all the application-specific details are defined. This problem class defines an API that the solver class applies for communication.

The solver class will usually have a function to set up data structures for the variational formulation, a `step` function to advance the solution one time step, and a `solve` function to run the time loop. Every time the solver class needs problem-specific information, it gets that information from the problem class, either in terms of attributes (variables) in the problem class or in terms of method (function) calls. The forthcoming examples are tied to the diffusion equation, but should be sufficiently general to be reused for most time-dependent FEniCS applications.

2.2.1 Mathematical problem

We address a variable-coefficient diffusion equation with Dirichlet, Neumann, and Robin conditions:

$$\varrho c\frac{\partial u}{\partial t} = \nabla\cdot(\kappa\nabla u) + f(\boldsymbol{x},t) \text{ in } \Omega\times(0,T], \tag{2.1}$$

$$u(\boldsymbol{x},0) = I \text{ on } \Omega, \tag{2.2}$$

$$u = u_{\mathrm{D}}(t) \text{ on } \Gamma_D, \tag{2.3}$$

$$-\kappa\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N, \tag{2.4}$$

$$-\kappa\frac{\partial u}{\partial n} = r(u-U_s) \text{ on } \Gamma_R\,. \tag{2.5}$$

The spatial domain Ω has boundary $\partial\Omega = \Gamma_D\cup\Gamma_N\cup\Gamma_R$. We shall assume that all coefficients ϱ, c, κ may vary in space, while f and g may vary in time too. The coefficients r and U_s are assumed to depend on time only.

We discretize in time by the general θ-rule. For an evolution equation $\partial P/\partial t = Q(t)$, this rule reads

$$\frac{P^{n+1}-P^n}{\Delta t} = \theta Q^{n+1} + (1-\theta)Q^n,$$

where $\theta\in[0,1]$ is a weighting factor. The attractive property of this scheme is that $\theta = 1$ corresponds to the Backward Euler scheme, $\theta = 1/2$ to the Crank-Nicolson scheme, and $\theta = 0$ to the Forward Euler scheme.

Introducing the θ-rule in our PDE results in

$$\varrho c\frac{u^{n+1}-u^n}{\Delta t} = \theta(\nabla\cdot\left(\kappa\nabla u^{n+1}\right)+f(\boldsymbol{x},t_{n+1}))+(1-\theta)(\nabla\cdot(\kappa\nabla u^n)+f(\boldsymbol{x},t_n))\,. \tag{2.6}$$

A Galerkin method for this initial-boundary value problem consists of multiplying (2.6) by a test function $v\in V$, integrate over Ω, and perform integration by parts on the second-order derivative term $\nabla\cdot(\kappa\nabla u)$:

$$\begin{aligned}\int_\Omega \big(v\varrho c\frac{u^{n+1}-u^n}{\Delta t}\,\mathrm{d}x &+ \theta\kappa\nabla u^{n+1}\cdot\nabla v - v\theta f(\boldsymbol{x},t_{n+1}\\ &+(1-\theta)\kappa\nabla u^n\cdot\nabla v - v(1-\theta)f(\boldsymbol{x},t_n)\big)\,\mathrm{d}x\\ &-\int_{\Gamma_N\cup\Gamma_R}\big(\theta\kappa\frac{\partial u^{n+1}}{\partial n}v+(1-\theta)\kappa\frac{\partial u^n}{\partial n}v\big)\,\mathrm{d}s = 0\,.\end{aligned}$$

Inserting the boundary conditions at Γ_N and Γ_R gives us

$$\begin{aligned}F(u;v) = \int_\Omega \big(v\varrho c\frac{u^{n+1}-u^n}{\Delta t}\,\mathrm{d}x &+ \theta\kappa\nabla u^{n+1}\cdot\nabla v - v\theta f(\boldsymbol{x},t_{n+1})\\ &-(1-\theta)\kappa\nabla u^n\cdot\nabla v + v(1-\theta)f(\boldsymbol{x},t_n)\big)\,\mathrm{d}x\\ &+\int_{\Gamma_N}\big(\theta g(\boldsymbol{x},t_{n+1})v+(1-\theta)g(\boldsymbol{x},t_n)v\big)\,\mathrm{d}s\\ &+\int_{\Gamma_R}\big(\theta r(u^{n+1}-U_s(t_{n+1}))v+(1-\theta)r(u^n-U_s(t_n))v\big)\,\mathrm{d}s = 0\,.\end{aligned} \tag{2.7}$$

Since we use `u` for the unknown u^{n+1} in the code, and `u_1` for u^n, we introduce the same notation in the mathematics too: u for u^{n+1} and u_1 for u^n,

$$\begin{aligned}F(u;v) = \int_\Omega \big(v\varrho c\frac{u-u_1}{\Delta t}\,\mathrm{d}x &+ \theta\kappa\nabla u\cdot\nabla v - v\theta f(\boldsymbol{x},t_{n+1})\\ &-(1-\theta)\kappa\nabla u_1\cdot\nabla v + v(1-\theta)f(\boldsymbol{x},t_n)\big)\,\mathrm{d}x\\ &+\int_{\Gamma_N}\big(\theta g(\boldsymbol{x},t_{n+1})v+(1-\theta)g(\boldsymbol{x},t_n)v\big)\,\mathrm{d}s\\ &+\int_{\Gamma_R}\big(\theta r(u-U_s(t_{n+1}))v+(1-\theta)r(u_1-U_s(t_n))v\big)\,\mathrm{d}s = 0\,.\end{aligned} \tag{2.8}$$

The variational formulation is then: at each time level, find $u \in V$ such that $F(u;v) = 0 \ \forall v \in V$. We do not need to identify the bilinear and linear terms in the expression F since we can use the `lhs` and `rhs` functions for this purpose in the code. However, we should be very convinced that we have a *linear* variational problem at hand and not a nonlinear one.

2.2.2 Superclass for problems

The solver class contains the data structures and actions from previous programs, but needs to ask the problem class about the mesh, boundary conditions, the time step, and so forth. We therefore need to define the API of the problem class first so we know how the solver class can ask for the mesh, for instance.

Here is an abstract problem class:

```
class DiffusionProblem(object):
    """Abstract base class for specific diffusion applications."""

    def solve(self, solver_class=DiffusionSolver,
              theta=0.5, linear_solver='direct',
              abs_tol=1E-6, rel_tol=1E-5, max_iter=1000):
        """Solve the PDE problem for the primary unknown."""
        self.solver = solver_class(self, theta)
        iterative_solver = KrylovSolver('gmres', 'ilu')
        prm = iterative_solver.parameters
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        prm['nonzero_initial_guess'] = True  # Use u (last sol.)
        return self.solver.solve()

    def flux(self):
        """Compute and return flux -p*grad(u)."""
        degree = self.solution().ufl_element().degree()
        V_g = VectorFunctionSpace(self.mesh, 'P', degree)
        flux_u = -self.heat_conduction()*grad(self.solution())
        self.flux_u = project(flux_u, V_g)
        self.flux_u.rename('flux(u)', 'continuous flux field')
        return self.flux_u

    def mesh_degree(self):
        """Return mesh, degree."""
        raise NotImplementedError('Must implement mesh')

    def I(self):
        """Return initial condition."""
        return Constant(0.0)

    def heat_conduction(self):  # kappa
```

```
        return Constant(1.0)

    def density(self):          # rho
        return Constant(1.0)

    def heat_capacity(self):    # c
        return Constant(1.0)

    def heat_source(self, t):   # f
        return Constant(0.0)

    def time_step(self, t):
        raise NotImplentedError('Must implement time_step')

    def end_time(self):
        raise NotImplentedError('Must implement end_time')

    def solution(self):
        return self.solver.u

    def user_action(self, t, u):
        """Post process solution u at time t."""
        pass

    def Dirichlet_conditions(self, t):
        """Return either an Expression (for the entire boundary) or
        a list of (value,boundary_parts,index) triplets."""
        return []

    def Neumann_conditions(self, t):
        """Return list of (g,ds(n)) pairs."""
        return []

    def Robin_conditions(self, t):
        """Return list of (r,s,ds(n)) triplets."""
        return []
```

The meaning of the different methods in this class will be evident as we present specific examples on implementations.

The idea now is that different problems are implemented as different subclasses of `DiffusionProblem`. The `solve` and `flux` methods are general and can be inherited, while the rest of the methods must be implemented in the subclass for the particular problem at hand.

2.2.3 A specific problem class

As a simple example, consider the test problem where we have a manufactured solution $u = 1 + x^2 + \alpha y^2 + \beta t$ on a uniform mesh over the unit square or cube,

with Dirichlet conditions on the entire boundary. Suppose we have $\Delta t = 0.3$ and want to simulate for $t \in [0, 0.9]$. A problem class is then

```
class TestProblemExact(DiffusionProblem):
    def __init__(self, Nx, Ny, Nz=None, degree=1, num_time_steps=3):
        if Nz is None:
            self.mesh = UnitSquareMesh(Nx, Ny)
        else:
            self.mesh = UnitCubeMesh(Nx, Ny, Nz)
        self.degree = degree
        self.num_time_steps = num_time_steps

        alpha = 3; beta = 1.2
        self.u0 = Expression(
            '1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
            alpha=alpha, beta=beta, t=0)
        self.f = Constant(beta - 2 - 2*alpha)

    def time_step(self, t):
        return 0.3

    def end_time(self):
        return self.num_time_steps*self.time_step(0)

    def mesh_degree(self):
        return self.mesh, self.degree

    def I(self):
        """Return initial condition."""
        return self.u0

    def heat_source(self, t):
        return self.f

    def Dirichlet_conditions(self, t):
        self.u0.t = t
        return self.u0

    def user_action(self, t, u, timestep):
        """Post process solution u at time t."""
        u_e = interpolate(self.u0, u.function_space())
        error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
        print('error at %g: %g' % (t, error))
        tol = 2E-11
        assert error < tol, 'max_error: %g' % error
```

Remember that we can inherit all methods from the parent class that are appropriate for the problem at hand.

Our test problem can now be solved in (e.g.) a unit test like

```
def test_DiffusionSolver():
    problem = TestProblemExact(Nx=2, Ny=2)
    problem.solve(theta=1, linear_solver='direct')
```

```
    u = problem.solution()
```

The solver class will call the `user_action` function at every time level, and this function will assert that we recover the solution to machine precision.

2.2.4 The solver class

The solver class, here based on the θ-rule and the variational formulation from the previous section, can be coded as follows:

```
class DiffusionSolver(object):
    """Solve a heat conduction problem by the theta-rule."""
    def __init__(self, problem, theta=0.5):
        self.problem = problem
        self.theta = theta

    def solve(self):
        """Run time loop."""
        tol = 1E-14
        T = self.problem.end_time()
        t = self.problem.time_step(0)
        self.initial_condition()
        timestep = 1

        while t <= T+tol:
            self.step(t)
            self.problem.user_action(t, self.u, timestep)

            # Updates
            self.dt = self.problem.time_step(
                t+self.problem.time_step(t))
            t += self.dt
            timestep += 1
            self.u_1.assign(self.u)

    def initial_condition(self):
        self.mesh, degree = self.problem.mesh_degree()
        self.V = V = FunctionSpace(self.mesh, 'P', degree)

        if hasattr(self.problem, 'I_project'):
            I_project = getattr(self.problem, 'I_project')
        else:
            I_project = False
        self.u_1 = project(self.problem.I(), V) if I_project \
                   else interpolate(self.problem.I(), V)
        self.u_1.rename('u', 'initial condition')
        self.u = self.u_1 # needed if flux is computed in the next step
        self.problem.user_action(0, self.u_1, 0)

    def step(self, t, linear_solver='direct',
             abs_tol=1E-6, rel_tol=1E-5, max_iter=1000):
```

```
        """Advance solution one time step."""
        # Find new Dirichlet conditions at this time step
        Dirichlet_cond = self.problem.Dirichlet_conditions(t)
        if isinstance(Dirichlet_cond, Expression):
            # Just one Expression for Dirichlet conditions on
            # the entire boundary
            self.bcs = [DirichletBC(
                self.V, Dirichlet_cond,
                lambda x, on_boundary: on_boundary)]
        else:
            # Boundary SubDomain markers
            self.bcs = [
                DirichletBC(self.V, value, boundaries, index)
                for value, boundaries, index
                in Dirichlet_cond]

        #debug_Dirichlet_conditions(self.bcs, self.mesh, self.V)

        self.define_variational_problem(t)
        a, L = lhs(self.F), rhs(self.F)
        A = assemble(a)
        b = assemble(L)

        [bc.apply(A, b) for bc in self.bcs]
        if self.V.dim() < 50:
            print('A:\n', A.array(), '\nb:\n', b.array())

        # Solve linear system
        if linear_solver == 'direct':
            solve(A, self.u.vector(), b)
        else:
            solver = KrylovSolver('gmres', 'ilu')
            solver.solve(A, self.u.vector(), b)

    def define_variational_problem(self, t):
        """Set up variational problem at time t."""
        u = TrialFunction(self.V)
        v = TestFunction(self.V)

        dt     = self.problem.time_step(t)
        kappa  = self.problem.heat_conduction()
        varrho = self.problem.density()
        c      = self.problem.heat_capacity()
        f      = self.problem.heat_source(t)
        f_1    = self.problem.heat_source(t-dt)

        theta = Constant(self.theta)
        u_1 = self.u_1  # first computed in initial_condition

        F = varrho*c*(u - u_1)/dt*v
        F += theta    *dot(kappa*grad(u),   grad(v))
        F += (1-theta)*dot(kappa*grad(u_1), grad(v))
        F -= theta*f*v + (1-theta)*f_1*v
        F = F*dx
```

```
        F += theta*sum(
            [g*v*ds_
             for g, ds_ in self.problem.Neumann_conditions(t)])
        F += (1-theta)*sum(
            [g*v*ds_
             for g, ds_ in self.problem.Neumann_conditions(t-dt)])
        F += theta*sum(
            [r*(u - U_s)*v*ds_
             for r, U_s, ds_ in self.problem.Robin_conditions(t)])
        F += (1-theta)*sum(
            [r*(u - U_s)*v*ds_
             for r, U_s, ds_ in self.problem.Robin_conditions(t-dt)])
        self.F = F

        self.u = Function(self.V)
        self.u.rename('u', 'solution')
```

2.3 Applications to heat conduction

We shall now through some real physical examples show how the problem classes can be constructed for various types of applications. The goal is to achieve PDE solvers that are flexible and convenient for performing scientific investigations.

2.3.1 Thermal boundary layer

Assume we have some medium at temperature U_s and then we suddenly heat one end so the temperature here stays constant at U_1. At the other end we have some equipment to keep the temperature constant at U_s. The other boundaries are insulated so heat cannot escape. There are no heat sources. How is the temperature development in the material due to such sudden heating of one end? Figure 2.1 sketches the situation (with a scaled variable u that jumps from 0 to 1).

Mathematics. The problem is mathematically one-dimensional, so it means that if we create a 2D or 3D domain, the boundaries in y and z directions are insulated (requiring $\partial u/\partial n = 0$ as boundary condition on $y = \text{const}$ and $z = \text{const}$). The heating is applied to $x = 0$ and $x = L$.

It is natural to scale the problem by introducing dimensionless independent and dependent variables:

$$\bar{x} = \frac{x}{L}, \quad \bar{y} = \frac{y}{L}, \quad \bar{u} = \frac{u - U_s}{U_1 - U_s}, \quad \bar{t} = \frac{t}{t_c}.$$

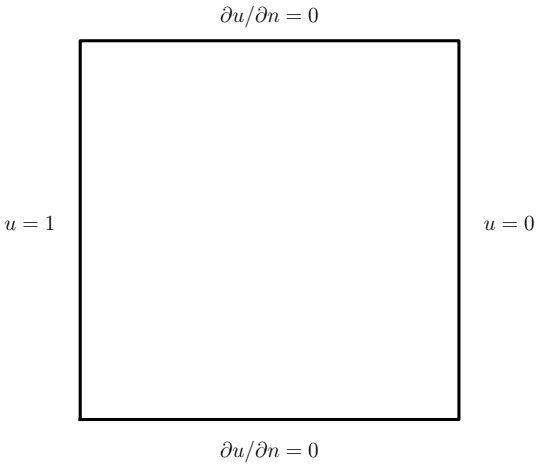

Fig. 2.1 Domain with (scaled) boundary conditions: sudden jump in u at the left boundary.

The suggested scaling for u makes a simple boundary condition at $x = 0$: $\bar{u} = 1$. This scaling also results in $\bar{u} \in [0,1]$ as is always desired.

After inserting the dimensionless variables in the PDE, we demand the time-derivative term and the heat conduction term to balance, and find t_c from that condition: $t_c = \varrho c L^2/\kappa$.

The spatial domain is the unit square. We introduce the boundaries Γ_{D_1} as the side $x = 0$, Γ_{D_2} as the side $x = 1$, and Γ_N as the rest of the boundary. The scaled initial-boundary problem can be written as

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \bar{\nabla}^2 \bar{u} \text{ in } \Omega = (0,1) \times (0,1) \times (0,T], \tag{2.9}$$

$$\bar{u}(\boldsymbol{x}, 0) = 0 \text{ in } \Omega, \tag{2.10}$$

$$\bar{u} = 1 \text{ at } \Gamma_{D_1}, \tag{2.11}$$

$$\bar{u} = 0 \text{ at } \Gamma_{D_2}, \tag{2.12}$$

$$\frac{\partial \bar{u}}{\partial \bar{n}} = 0 \text{ at } \Gamma_N . \tag{2.13}$$

FEniCS implementation. We can solve our problem with the general problem and solver classes by setting $\varrho = c = \kappa = 1$, and $I = 0$. The most labor-intensive part of the problem class is the visualization. We can create a helper class, `ProcessSolution`, which applies `cbcpost` to store the solution and perform animation via the `fenics.plot` tool:

```
import cbcpost as post
class ProcessSolution(object):
    """user_action function for storing the solution and flux."""
```

```
    def __init__(self, problem, u_min=0, u_max=1):
        """Define fields to be stored/plotted."""
        self.problem = problem  # this user_action belongs to problem
        self.pp = post.PostProcessor(
            dict(casedir='Results', clean_casedir=True))

        self.pp.add_field(
            post.SolutionField(
                'Solution',
                dict(save=True,
                     save_as=['hdf5', 'xdmf'],  # format
                     plot=True,
                     plot_args=
                     dict(range_min=float(u_min),
                          range_max=float(u_max))
                     )))

        self.pp.add_field(
            post.SolutionField(
                "Flux",
                dict(save=True,
                     save_as=["hdf5", "xdmf"],  # format
                     )))

    def __call__(self, t, u, timestep):
        """Store u and its flux to file."""
        u.rename('u', 'Solution')
        self.pp.update_all(
            {'Solution': lambda: u,
             'Flux': lambda: self.problem.flux()},
            t, timestep)
        info('saving results at time %g, max u: %g' %
             (t, u.vector().array().max()))
```

In the `user_action` method, we use this tool to store the solution, but we also add statements for plotting u along a line from $x = 0$ to $x = 1$ through the medium ($y = 0.5$). This gives an animation of the temperature profile, but results in somewhat lengthy code.

To mark the boundaries, so we can set $u = 1$ at $x = 0$, we can make a function like `mark_boundaries_in_hypercube` as shown in Section 1.1.4. Eventually, we are in a position to show the complete problem class:

```
class Problem1(DiffusionProblem):
    """Evolving boundary layer, I=0, but u=1 at x=0."""
    def __init__(self, Nx, Ny):
        self.make_mesh(self, Nx, Ny)
        # Storage and visualization
        self.user_action_object = \
                  ProcessSolution(self, u_min=0, u_max=1)
        # Compare u(x,t) as curve plots for the following times
        self.times4curveplots = [
            self.time_step(0), 4*self.time_step(0),
            8*self.time_step(0), 12*self.time_step(0),
```

```
            16*self.time_step(0), 0.02, 0.1, 0.2, 0.3]
        plt.ion()  # for animation

    def make_mesh(self, Nx, Ny):
        """Initialize mesh, boundary parts, and p."""
        print('XXX in Problem1.make_mesh')
        self.mesh = UnitSquareMesh(Nx, Ny)
        self.divisions = (Nx, Ny)

        self.boundary_parts = \
             mark_boundaries_in_hypercube(self.mesh, d=2)
        self.ds =  Measure(
            'ds', domain=self.mesh,
            subdomain_data=self.boundary_parts)

    def time_step(self, t):
        # Small time steps in the beginning when the boundary
        # layer develops
        if t < 0.02:
            return 0.0005
        else:
            return 0.025

    def end_time(self):
        return 0.3

    def mesh_degree(self):
        return self.mesh, 1

    def Dirichlet_conditions(self, t):
        """Return list of (value,boundary) pairs."""
        return [(1.0, self.boundary_parts, 0),
                (0.0, self.boundary_parts, 1)]

    def user_action(self, t, u, timestep):
        """Post process solution u at time t."""
        tol = 1E-14
        self.user_action_object(t, u, timestep)
        # Also plot u along line y=1/2
        x_coor = np.linspace(tol, 1-tol, 101)
        x = [(x_,0.5) for x_ in x_coor]
        u = self.solution()
        u_line = [u(x_) for x_ in x]
        # Animation in figure(1)
        plt.figure(1)
        if timestep == 0:
            self.lines = plt.plot(x_coor, u_line, 'b-')
            plt.legend(['u, t=%.4f' % t])
            plt.title('Solution along y=1/2, time step: %g' %
                      self.time_step(t))
            plt.xlabel('x'), plt.ylabel('u')
            plt.axis([0, 1, 0, 1])
            plt.savefig('tmp_%04d.png' % timestep)
        else:
```

```
            self.lines[0].set_ydata(u_line)
            plt.title('Solution along y=1/2, time step: %g' %
                      self.time_step(t))
            plt.draw()
            plt.savefig('tmp_%04d.png' % timestep)
        # Accumulated selected curves in one plot in figure(2)
        plt.figure(2)
        for t_ in self.times4curveplots:
            if abs(t - t_) < 0.5*self.time_step(t):
                plt.plot(x_coor, u_line, '-')
                plt.legend(['u, t=%.4f' % t])
                plt.xlabel('x'); plt.ylabel('u')
                plt.axis([0, 1, 0, 1])
                plt.hold('on')
```

Notice our definition of the time step: because the growth of the thin boundary layer close to $x = 0$ is very rapid for small times, we need to start with a small time step. Nevertheless, the speed of the heat transfer slows down with time, so we decide to use a longer time step after $t = 0.02$. The animation would otherwise also be boring to watch, but be aware of the fact that the apparent speed of the physical process is dramatically increased in the animation at $t = 0.02$.

The problem is solved by

```
def demo_Problem1():
    problem = Problem1(Nx=20, Ny=5)
    problem.solve(theta=1, linear_solver='direct')
    plt.figure(2)
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')
```

Results. Figure 2.2 shows accumulated curves (from `plt.figure(2)`). The problem is a primary example on a *thermal boundary layer*: the sudden rise in temperature at $x = 0$ at $t = 0$ gives rise to a very steep function, and a thin boundary layer that grows with time as heat is transported from the boundary into the domain. The jump in the temperature profile at $x = 0$ makes demands to the numerical methods. Quite typically, a Crank-Nicolson scheme may show oscillations (as we can see in the first curve) because of inaccurate treatment of the shortest spatial waves in the Fourier representation of the discrete solution. The oscillations are removed by doubling the spatial resolution from 20 to 40 elements in the x direction. With $\theta = 1$, we never experience any oscillations, but the boundary layer gets thicker and less accurate (smaller Δt is needed to compensate). However, in this problem, we see from Figure 2.2 that the inaccuracy is only visible for the very first time steps as the boundary layer is thin.

From all the plot frames with filenames `tmp_%04d.png` we may create video files by

Terminal

```
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec libx264   movie.mp4
Terminal> ffmpeg -i tmp_%04d.png -r 25 -vcodec libtheora movie.ogg
```

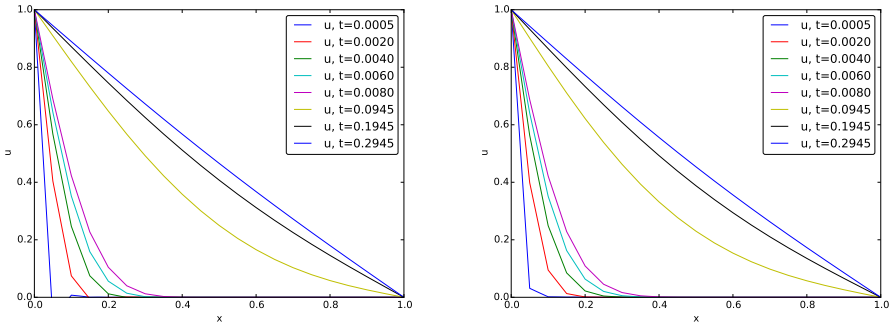

Fig. 2.2 Development of thermal boundary layer: Crank-Nicolson (left) and Backward Euler (right) schemes.

Movie 2: Developing thermal boundary layer (notice the jump in speed, i.e., time step!) `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/thermal_layer1/movie.ogg`

2.3.2 Extension to a heterogeneous medium

Suppose we now place another material inside the domain with other values material properties (i.e., values of ϱ, c, and κ). The new material occupies the rectangle $[0.3, 0.7] \times [0.3, 0.7]$ inside the scaled domain. We also change the boundary condition at $x = 1$ to be "no change", i.e., $\partial u/\partial n = 0$. Figure 2.3 depicts the problem.

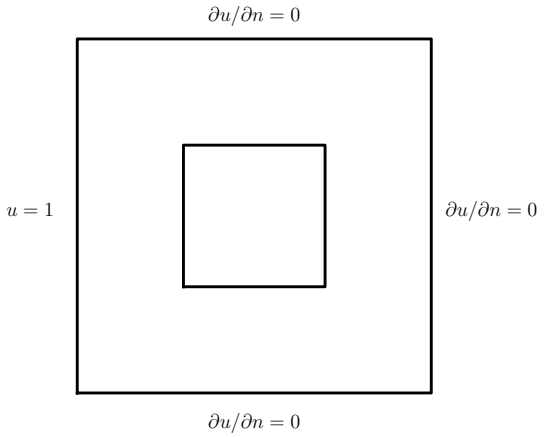

Fig. 2.3 Domain with internal subdomain and (scaled) boundary conditions.

Updated scaling. The former scaling is not completely valid as it was based on constant ϱ, c, and κ. We now introduce

$$\bar{\varrho} = \frac{\varrho}{\varrho_0}, \quad \bar{c} = \frac{c}{c_0}, \quad \bar{\kappa} = \frac{\kappa}{\kappa_0},$$

where ϱ_0 is the value of ϱ in the outer material, now to be known as subdomain 0. A similar parameter ϱ_1 is the value of ϱ inside the new material, called subdomain 1. The constants c_0, κ_0, c_1, and κ_1 are defined similarly. In subdomain 0, $\bar{\varrho} = 1$, and in subdomain 1, $\bar{\varrho} = \varrho_1/\varrho_0$, with similar values for $\bar{c}$ and $\bar{\kappa}$. The scaled PDE becomes

$$\bar{\varrho}\bar{c}\frac{\partial \bar{u}}{\partial \bar{t}} = \bar{\nabla}\cdot(\bar{\kappa}\bar{\nabla}\bar{u}) + \bar{f}\,.$$

We can call up the solver for the problem with dimensions as long as we remember to set $\kappa = \varrho = c = 1$ in subdomain 0. In subdomain 1, we divide by $\bar{\varrho} = \varrho_1/\varrho_0$ and $\bar{c} = c_1/c_0$, which results in a coefficient

$$\alpha = \frac{\varrho_0 c_0 \kappa_1}{\varrho_1 c_1 \kappa_0}$$

on the right-hand side. This means that we can let `density` and `heat_capacity` be of unit value and only operate with a spatially varying κ, which takes on the values 1 in subdomain 0 and α in subdomain 1. For simplicity, we just name this parameter `kappa_values` in the code.

hpl 9: Is this trick too tricky? Would it be clearer to let all three parameters vary?

The problem class. The problem class is very similar to `Problem1` above, except for the fact that we need to define the inner subdomain, we need to allow for κ values in subdomain 0 and 1, the time points for plots and time steps are a bit different, and the Dirichlet condition only applies to $x = 0$ (no need to implement the Neumann condition as long as it is zero).

```
class Problem2(Problem1):
    """As Problem 1, but du/dn at x=1 and varying kappa."""
    def __init__(self, Nx, Ny, kappa_values, subdomain_corners):
        """
        Nx x Ny mesh. kappa_values=[1,a],
        subdomain_corners=[(0.3,0.3), (0.7,0.9)]
        """
        self.make_mesh(Nx, Ny, kappa_values, subdomain_corners)
        self.user_action_object = \
                  ProcessSolution(self, u_min=0, u_max=1)
        # Compare u(x,t) as curve plots for the following times
        self.times4curveplots = [
            12*self.time_step(0),
            0.02, 0.1, 0.3, 0.5]
        plt.ion()  # for animation
```

```
    def make_mesh(self, Nx, Ny, kappa_values, subdomain_corners):
        """Initialize mesh, boundary parts, and p."""
        self.mesh = UnitSquareMesh(Nx, Ny)
        self.divisions = (Nx, Ny)

        self.boundary_parts = \
             mark_boundaries_in_hypercube(self.mesh, d=2)
        self.ds =  Measure(
            'ds', domain=self.mesh,
            subdomain_data=self.boundary_parts)

        # The domain is the unit square with an embedded rectangle
        class Rectangle(SubDomain):
            def __init__(self, subdomain_def):
                self.subdomain_def = subdomain_def
                SubDomain.__init__(self)

            def inside(self, x, on_boundary):
                # subdomain_def:
                # 0.3 <= x[0] <= 0.7 && 0.5 <= x[1] <= 0.9
                return eval(self.subdomain_def)

        self.materials = CellFunction('size_t', self.mesh)
        self.materials.set_all(0)  # "the rest"
        # Give subdomain_corners as (c,c),(b,b)
        c = subdomain_corners
        subdomain_str = '%g <= x[0] <= %g and %g <= x[1] <= %g' % \
                        (c[0][0], c[1][0], c[0][1], c[1][1])
        #subdomain = CompiledSubDomain(subdomain_str.replace('and', '&&'))
        subdomain = Rectangle(subdomain_str)
        subdomain.mark(self.materials, 1)
        self.V0 = FunctionSpace(self.mesh, 'DG', 0)
        self.kappa = Function(self.V0)
        help = np.asarray(self.materials.array(), dtype=np.int32)
        self.kappa.vector()[:] = np.choose(help, kappa_values)
        plot(self.materials, title='Subdomain', interactive=True)

    def time_step(self, t):
        if t < 0.04:
            return 0.0005
        else:
            return 0.025

    def end_time(self):
        return 0.5

    def heat_conduction(self):
        return self.kappa

    def Dirichlet_conditions(self, t):
        """Return list of (value,boundary) pairs."""
        return [(1.0, self.boundary_parts, 0)]
```

Results. We run a case where $\alpha = 1000$:

```
def demo_Problem2():
    problem = Problem2(Nx=20, Ny=5, kappa_values=[1,1000],
              subdomain_corners=[(0.3,0.3), (0.7,0.7)])
    problem.solve(theta=0.5, linear_solver='direct')
    plt.figure(2)
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')
```

As shown in Figure 2.4, the highly conductive inner material leads to a flat temperature profile in this region. The start of the process is as before, but with an insulated boundary at $x = 1$, heat builds up with time. The limiting steady state is $u = 1$ as $t \to \infty$.

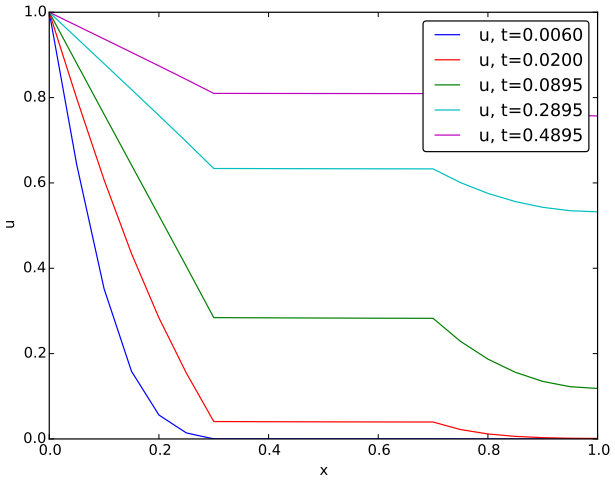

Fig. 2.4 Development of thermal boundary layer in heterogeneous medium.

Movie 3: Developing thermal boundary layer in heterogeneous medium (notice the jump in speed, i.e., time step!) `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/thermal_layer2/movie.ogg`

2.3.3 Oscillating boundary temperature

The next example concerns the question: How is the temperature in the ground affected by day and night variations at the earth's surface? We consider a rectangular domain with an embedded subdomain as in the previous example. At the side $y = 1$ (representing the earth's surface), we have an oscillating temperature:

$$u_B(t) = U_s + A\sin(wt),$$

where U_s is the mean temperature, $[-A, A]$ is the temperature variation, and w is the frequency, here equal to $w = 2\pi/P$, where P is the period of 24 h.

At the other boundaries we assume symmetry or "no change", which implies $\partial u/\partial n = 0$. The initial condition is taken as $u = U_s$, but any value will be lost in long time simulations as a steady state oscillatory condition is established. Figure 2.5 shows the domain and boundary conditions.

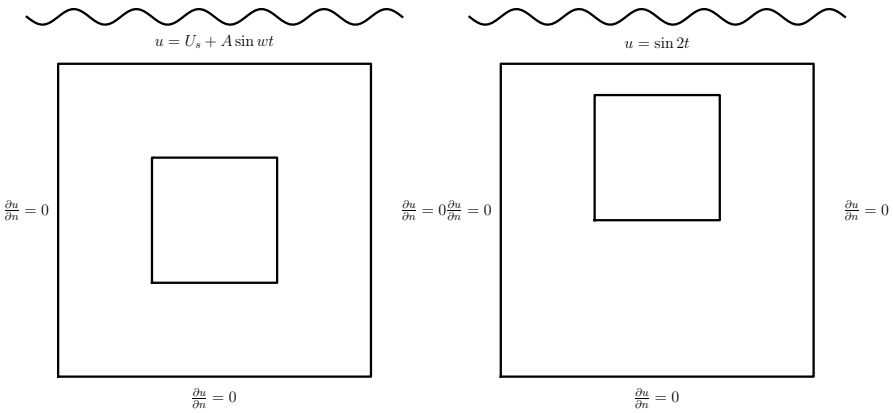

Fig. 2.5 Domain with oscillating temperature at the boundary: unscaled (left) and scaled (left).

Scaling. Now we expect u to oscillate around U_s with amplitude A, so to have $\bar{u} \in [-1, 1]$, we set

$$\bar{u} = \frac{u - U_s}{A}.$$

The scaled boundary condition is then

$$\bar{u}_B(\bar{t}) = \sin(wt_c\bar{t}).$$

We use a time scale based on w, i.e., $t_c = 1/w$. Chapter 3.2.4 in [3] (see ebook[1]) has an in-depth coverage of the scaling of this problem. The challenge is that the temperature will oscillate close to $y = 1$, but the oscillations will decay as we move downwards. One can for special set of parameters get very thin oscillating boundary layers, which make great demands to the numerical methods, or one may not achieve substantial decay so the boundary condition on $y = 0$ becomes wrong. To zoom in on the solution in the right way, it turns out that the right spatial length scale is $\sqrt{2\kappa/(wc\varrho)}$. With this length scale,

[1] `http://hplgit.github.io/scaling-book/doc/pub/book/html/._scaling-book008.html#___sec142`

a typical length of the domain in y direction is 4. The most appealing time scale is $t_c = 2/w$.

We end up with the same scaled problem as in the previous section, except that at $y = 1$ we have

$$\bar{u}_B(\bar{t}) = \sin(2\bar{t})\,.$$

The problem class. We need a different reasoning about the time steps size since this is an oscillatory problem. We also need to stretch the unit square so it becomes $[0,4] \times [0,4]$ as desired. In addition, we need to change the Dirichlet condition. And finally, we need to adjust the curve plotting as it now takes place in y direction, and the axes are different. Much of class `Problem2` can be reused, so it makes sense to make a subclass and override the methods that do not fit.

```
class Problem3(Problem2):
    """Oscillating surface temperature."""
    def __init__(self, Nx, Ny, kappa_values, subdomain_corners):
        # Oscillating temperature at x=0:
        self.surface_temp = lambda t: sin(2*t)
        w = 2.0
        period = 2*np.pi/w
        self.dt = period/30  # need this before Problem2.__init__
        self.T = 4*period

        Problem2.__init__(
            self, Nx, Ny, kappa_values, subdomain_corners)
        # Stretch the mesh in y direction so we get [0,4]x[0,4]
        self.mesh.coordinates()[:] *= 4

        self.user_action_object = \
                  ProcessSolution(self, u_min=-1, u_max=1)
        # Compare u(x,t) as curve plots for the following times
        self.times4curveplots = [
            period/4, 4*period/8, 3*period/4, 3*period]

    def time_step(self, t):
        return self.dt

    def end_time(self):
        return self.T

    def Dirichlet_conditions(self, t):
        """Return list of (value,boundary) pairs."""
        return [(self.surface_temp(t), self.boundary_parts, 3)]

    def user_action(self, t, u, timestep):
        """Post process solution u at time t."""
        tol = 1E-14
        self.user_action_object(t, u, timestep)
        # Also plot u along line x=2
        y_coor = np.linspace(tol, 4-tol, 101)
```

```
        y = [(2, y_) for y_ in y_coor]
        u = self.solution()
        u_line = [u(y_) for y_ in y]
        # Animation in figure(1)
        plt.figure(1)
        if timestep == 0:
            self.lines = plt.plot(y_coor, u_line, 'b-')
            plt.legend(['u, t=%.4f' % t])
            plt.title('Solution along x=2, time step: %g' %
                      self.time_step(t))
            plt.xlabel('y'); plt.ylabel('u')
            plt.axis([0, 4, -1.2, 1.2])
            plt.savefig('tmp_%04d.png' % timestep)
        else:
            self.lines[0].set_ydata(u_line)
            plt.title('Solution along x=2, time step: %g' %
                      self.time_step(t))
            plt.draw()
            plt.savefig('tmp_%04d.png' % timestep)

        # Accumulated selected curves in one plot in figure(2)
        plt.figure(2)
        for t_ in self.times4curveplots:
            if abs(t - t_) < 0.5*self.time_step(t):
                plt.plot(y_coor, u_line, '-')
                plt.legend(['u, t=%.4f' % t])
                plt.xlabel('y'); plt.ylabel('u')
                plt.axis([0, 4, -1.2, 1.2])
                plt.hold('on')
```

The problem is solved by

```
def demo_Problem3():
    problem = Problem3(Nx=5, Ny=20, kappa_values=[1, 1000],
              subdomain_corners=[(0.3,0.4), (0.7,0.8)])
    problem.solve(theta=0.5, linear_solver='direct')
    plt.figure(2)
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')
```

Results. We have made runs with a homogeneous medium and with a heterogeneous medium (using $\alpha = 1000$ as in the previous section). Animation in ParaView meets the problem that $u = \text{const}$ initially so we must manually set a range for the data. Bring up the Color Map Editor (click on **Edit** in the *Coloring* section in the left part of the GUI), click on the second icon from the top, to "rescale the custom range", give -1 and 1 as the data range, and click **Update** to bring this range into action.

Movie 4: Oscillating boundary temperature and homogeneous medium. `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/thermal_layer3/movie.ogg`

Movie 5: Oscillating boundary temperature and heterogeneous medium. `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/thermal_layer4/movie.ogg`

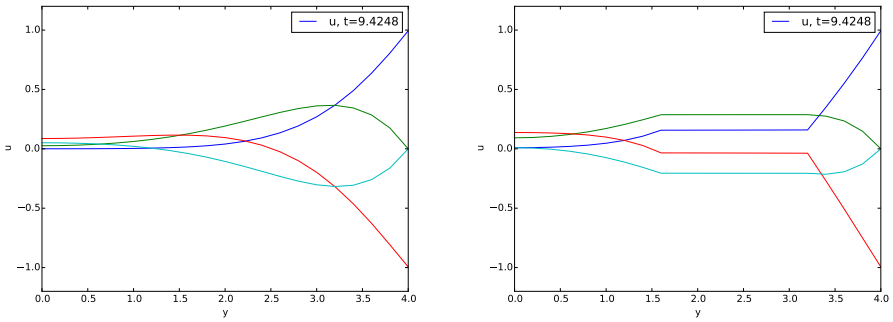

Fig. 2.6 Oscillating boundary temperature: homogeneous (left) and heterogeneous (right) medium.

Movie 6: Scalar field animation (homogeneous medium). `https://raw.githubusercontent.com/hplgit/fenics-tutorial/brief/doc/src/mov/thermal_layer3/paraview.ogg`

Tip: Let related problem classes utilize inheritance

The last three examples regard quite related problems, yet they have substantial differences. The typical approach to making FEniCS software to these applications would be to have three flat programs, each containing a full solver of the PDE, but with details adapted to the problem at hand. The class approach, on the other hand, shows how all applications share the same numerical implementation. The different problem classes can also share a lot of code so inheritance is a way to save writing. However, such class programming requires some experience as it is easy to make mistakes and inherit functionality that is wrong.

Chapter 3
Implementing solvers for nonlinear PDEs

FEniCS almost automates the solution of nonlinear problems, as shown in Section 3.2 in [2]. However, in problems with severe nonlinearities it can be hard to obtain convergence of the iterative methods, and one needs to tailor the solution to the problem at hand. FEniCS is very well suited for this, but then the nonlinear iteration algorithm must be hand-coded. This chapter shows the implementation of three basic solution strategies:

- a simple Picard-type iteration
- a Newton method at the algebraic level
- a Newton method at the PDE level

The detailed exposition will hopefully also help newcomers to the finite element method to understand the method better.

Test problem. As basic test problem in this chapter, we use the nonlinear Poisson problem

$$-\nabla\cdot(q(u)\nabla u) = f\,.$$

It turns out that

$$u(x_0,\ldots,x_{d-1}) = \left((2^{m+1}-1)x_0+1\right)^{1/(m+1)} - 1 \tag{3.1}$$

is an exact solution of this nonlinear PDE if $f=0$, $q(u)=(1+u)^m$, the domain Ω is the unit hypercube $[0,1]^d$ in d dimensions, $u=0$ for $x_0=0$, $u=1$ for $x_0=1$, and $\partial u/\partial n=0$ at all other boundaries $x_i=0$ and $x_i=1$, $i=1,\ldots,d-1$. The coordinates are now represented by the symbols $x_0,\ldots,x_{d-1}$.

We refer to Section 6.1.5 in [2] for details on formulating a PDE problem in d space dimensions.

3.1 The built-in automated Newton solver

Solving nonlinear problems in FEniCS is a matter of defining the nonlinear variational form `F` and calling `solve(F == 0, ...)` as explained in Section 3.2.3 in [2]. However, in real problems we will at least have control of the tolerance used in the stopping criterion in Newton's method. Sometimes we also want to derive the Jacobian ourselves or provide approximations. Such customizations are shown next.

3.1.1 Computing the Jacobian

We may provide the Jacobian for Newton's method as argument to `solve`: `solve(F == 0, u, bcs=bcs, J=J)`. However, if you have several nonlinear problems within the same code and want different solvers for them, it is better to use these objects:

```
problem = NonlinearVariationalProblem(F, u, bcs, J)
solver  = NonlinearVariationalSolver(problem)
solver.solve()
```

Here, `F` corresponds to the nonlinear form $F(u;v)$, `u` is the unknown `Function` object, `bcs` represents the essential boundary conditions (in general a list of `DirichletBC` objects), and `J` is a variational form for the Jacobian of `F`.

The `F` form corresponding to

$$F(u;v) = \int_\Omega q(u)\nabla u\cdot\nabla v + fv\,\mathrm{d}x, \tag{3.2}$$

is straightforwardly defined as follows, assuming `q(u)` is coded as a Python function (see Section 3.2.3 in [2]) for code):

```
u_ = Function(V)     # most recently computed solution
v  = TestFunction(V)
F  = dot(q(u_)*grad(u_), grad(v))*dx
```

Note here that `u_` is a `Function` and not a `TrialFunction`. An alternative and perhaps more intuitive formula for F is to define $F(u;v)$ directly in terms of a trial function for u and a test function for v, and then create the proper `F` by

```
u  = TrialFunction(V)
v  = TestFunction(V)
F  = dot(q(u)*grad(u), grad(v))*dx
u_ = Function(V)     # the most recently computed solution
F  = action(F, u_)
```

The latter statement is equivalent to $F(u=u^-;v)$, where u^- is an existing finite element function representing the most recently computed approximation

to the solution. (Note that u^k and u^{k+1} in the previous notation correspond to u^- and u in the present notation. We have changed notation to better align the mathematics with the associated code.)

The derivative J (`J`) of F (`F`) is formally the Gateaux derivative $DF(u^k;\delta u,v)$ of $F(u;v)$ at $u=u^-$ in the direction of δu. Technically, this Gateaux derivative is derived by computing

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u^- + \epsilon\delta u;v)\,. \tag{3.3}$$

The δu is now the trial function and u^- is the previous approximation to the solution u. We start with

$$\frac{d}{d\epsilon}\int_\Omega \nabla v\cdot\big(q(u^- + \epsilon\delta u)\nabla(u^- + \epsilon\delta u)\big)\,\mathrm{d}x$$

and obtain

$$\int_\Omega \nabla v\cdot\big[q'(u^- + \epsilon\delta u)\delta u\nabla(u^- + \epsilon\delta u) + q(u^- + \epsilon\delta u)\nabla\delta u\big]\,\mathrm{d}x,$$

which leads to

$$\int_\Omega \nabla v\cdot\big[q'(u^-)\delta u\nabla(u^-) + q(u^-)\nabla\delta u\big]\,\mathrm{d}x, \tag{3.4}$$

as $\epsilon\to 0$. This last expression is the Gateaux derivative of F. We may use J or $a(\delta u,v)$ for this derivative. The latter has the advantage of being easy to recognize as a bilinear form. However, in the forthcoming code examples `J` is used as variable name for the Jacobian.

The specification of `J` goes as follows if `du` is the `TrialFunction`:

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = dot(q(u_)*grad(u_), grad(v))*dx

J = dot(q(u_)*grad(du), grad(v))*dx + \
    dot(Dq(u_)*du*grad(u_), grad(v))*dx
```

With the alternative specification of `F`, where `u` is a `TrialFunction`, `J` is computed by

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = dot(q(u)*grad(u), grad(v))*dx
F  = action(F, u_)

J = dot(q(u_)*grad(u), grad(v))*dx + \
    dot(Dq(u_)*u*grad(u_), grad(v))*dx
```

The UFL language, used to specify weak forms, supports differentiation of forms. This feature facilitates automatic *symbolic* computation of the Jacobian `J` by calling the function `derivative` with `F`, the most recently computed solution (`Function`), and the unknown (`TrialFunction`) as parameters:

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = dot(q(u_)*grad(u_), grad(v))*dx

J  = derivative(F, u_, du)  # Gateaux derivative in dir. of du
```

or

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = dot(q(u)*grad(u), grad(v))*dx
F  = action(F, u_)

J  = derivative(F, u_, u)   # Gateaux derivative in dir. of u
```

The `derivative` function is obviously very convenient in problems where differentiating `F` by hand implies lengthy calculations.

3.1.2 Setting solver parameters

The following code defines the nonlinear variational problem and an associated solver based on Newton's method. We here demonstrate how key parameters in Newton's method can be set, as well as the choice of solver and preconditioner, and associated parameters, for the linear system occurring in the Newton iterations.

```
problem = NonlinearVariationalProblem(F, u_, bcs, J)
solver  = NonlinearVariationalSolver(problem)
prm = solver.parameters
prm_n = prm['newton_solver']
prm_n['absolute_tolerance'] = abs_tol_Newton
prm_n['relative_tolerance'] = rel_tol_Newton
prm_n['maximum_iterations'] = max_iter_Newton
prm_n['relaxation_parameter'] = relaxation_prm_Newton
if linear_solver == 'Krylov':
    prec = 'jacobi' if 'jacobi' in \
           list(zip(*krylov_solver_preconditioners()))[0] \
           else 'ilu'
    prm_n['linear_solver'] = 'gmres'
    prm_n['preconditioner'] = prec
    prm_nk = prm_n['krylov_solver']
    prm_nk['absolute_tolerance'] = abs_tol_Krylov
    prm_nk['relative_tolerance'] = rel_tol_Krylov
```

```
    prm_nk['maximum_iterations'] = max_iter_Krylov
    prm_nk['monitor_convergence'] = True
    prm_nk['nonzero_initial_guess'] = False
    prm_nk['gmres']['restart'] = 40
    prm_nk['preconditioner']['structure'] = \
                                    'same_nonzero_pattern'
    prm_nk['preconditioner']['ilu']['fill_level'] = 0
```

A list of available parameters and their default values can as usual be printed by calling `info(prm, True)`. The `u_` we feed to the nonlinear variational problem object is filled with the solution by the call `solver.solve()`.

3.1.3 FEniCS implementation

The preferred implementation of `F` and `J`, depending on whether `du` or `u` is the `TrialFunction` object, is a matter of personal taste. Derivation of the Gateaux derivative by hand, as shown above, is most naturally matched by an implementation where `du` is the `TrialFunction`, while use of automatic symbolic differentiation with the aid of the `derivative` function is most naturally matched by an implementation where `u` is the `TrialFunction`. We have implemented both approaches in a `solver` function in the file `ft13_nlpoisson_func.py`. An argument `TrialFunction_object` can be set to `u` if we want to have `u` as `TrialFunction`, otherwise it set to `du`.

```
def solver(
    q, Dq, f, divisions, degree=1,
    method='u', J_comp='manual',
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol_Krylov=1E-5,
    rel_tol_Krylov=1E-5,
    abs_tol_Newton=1E-5,
    rel_tol_Newton=1E-5,
    max_iter_Krylov=1000,
    max_iter_Newton=50,
    relaxation_prm_Newton=1.0,
    log_level=PROGRESS,
    dump_parameters=False,
    ):
```

Let us run the code for $m = 2$ and three meshes corresponding to `divisions` as `(10, 10)`, `(20, 20)`, and `(40, 40)`. The maximum errors are then $5 \cdot 10^{-3}$, $1.7 \cdot 10^{-3} \approx \frac{1}{4} 5 \cdot 10^{-3}$, $4.5 \cdot 10^{-4} \approx \frac{1}{4} 1.7 \cdot 10^{-3}$, and $1.2 \cdot 10^{-4} \approx \frac{1}{4} 4.5 \cdot 10^{-4}$, demonstrating second-order convergence in the cell size as expected for P1 elements. This test is turned into a proper unit test in the `test_solver` function.

3.2 Manual implementation of solution algorithms

We now show how we can implement the solution algorithm for handling nonlinear PDEs from scratch. First, we treat the common and popular Picard iteration method. Second, we look at a standard Newton method for solving nonlinear algebraic equations. And third, we demonstrate application of the Newton method directly to the PDE, which is an approach that is more attractive to FEniCS programmers than the second one.

3.2.1 Picard iteration

Idea. Picard iteration, also called the method of successive substitutions, is an easy way of handling nonlinear PDEs: we simply use a known, previous solution in the nonlinear terms so that these terms become linear in the unknown u. The strategy is also known as the method of successive substitutions. For our particular problem, we use a known, previous solution in the coefficient $q(u)$. More precisely, given a solution u^k from iteration k, we seek a new (hopefully improved) solution u^{k+1} in iteration $k+1$ such that u^{k+1} solves the *linear problem*,

$$\nabla\cdot\left(q(u^k)\nabla u^{k+1}\right)=0,\quad k=0,1,\ldots \tag{3.5}$$

The iterations require an initial guess u^0. The hope is that $u^k\to u$ as $k\to\infty$, and that u^{k+1} is sufficiently close to the exact solution u of the discrete problem after just a few iterations.

A similar linearization is needed in Neumann and Robin conditions as well, e.g., the condition

$$-q(u)\frac{\partial u}{\partial n}=r(u-U_s)$$

is linearized to

$$-q(u^k)\frac{\partial u^{k+1}}{\partial n}=r(u^{k+1}-U_s)\,.$$

Variational formulation. We can easily formulate a variational problem for u^{k+1} from (3.5) by the same steps as for a linear Poisson problem. The problem boils down to seeking $u^{k+1}\in V$ such that

$$\tilde{F}(u^{k+1};v)=0\quad\forall v\in\hat{V},\quad k=0,1,\ldots, \tag{3.6}$$

with

$$\tilde{F}(u^{k+1};v) = \int_\Omega q(u^k)\nabla u^{k+1}\cdot\nabla v\,\mathrm{d}x\,. \tag{3.7}$$

Since this is a linear problem in the unknown u^{k+1}, we can equivalently use the formulation

$$a(u^{k+1},v) = L(v), \tag{3.8}$$

with $a(u,v) = \int_\Omega q(u^k)\nabla u\cdot\nabla v\,\mathrm{d}x$ and $L(v) = 0$.

Stopping criteria. The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| <$ tol, where tol is a small tolerance, say 10^{-5}, or when the number of iterations exceed some critical limit. The latter case will pick up divergence of the method or unacceptable slow convergence. The residual criterion $||\epsilon_F \equiv ||F(u^{k+1};v)|| <$ tol can also be used, but it would require a kind of extra iteration since we must compute the linear system in the next iteration to be in position to evaluate $||F(u^{k+1};v)||$. However, since the residual criterion is costly, we use the change in solution ϵ instead.

FEniCS implementation. In the solution algorithm we only need to store u^k and u^{k+1}, called `u_` and `u` in the code below (our convention is that `u` is always the unknown to be computed in a code, and `u_` is the most recently computed approximation to the mathematical solution of the problem). The algorithm can then be expressed as follows:

```
def q(u):
    return (1+u)**m

# Define variational problem for Picard iteration
u = TrialFunction(V)
v = TestFunction(V)
u_ = interpolate(Constant(0.0), V)  # previous (known) u
a = dot(q(u_)*grad(u), grad(v))*dx
f = Constant(0.0)
L = f*v*dx

# Picard iterations
u = Function(V)     # new unknown function
eps = 1.0           # error measure ||u-u_||
tol = 1.0E-5        # tolerance
iter = 0            # iteration counter
maxiter = 25        # max no of iterations allowed
while eps > tol and iter < maxiter:
    iter += 1
    solve(a == L, u, bcs)
    du = u.vector().array() - u_.vector().array()
    eps = np.linalg.norm(du, ord=numpy.Inf)
    print('iter=%d: norm=%g' % (iter, eps))
    u_.assign(u)   # update for next iteration
```

We need to define the previous solution in the iterations, `u_`, as a finite element function so that `u_` can be updated with `u` at the end of the loop.

We may create the initial `Function u_` by interpolating an `Expression` or a `Constant` to the same vector space as `u` lives in (`V`).

In the code above we demonstrate how to use `numpy` functionality to compute the norm of the difference between the two most recent solutions. Here we apply the maximum norm (ℓ_∞ norm) on the difference of the solution vectors (`ord=1` and `ord=2` give the ℓ_1 and ℓ_2 vector norms - other norms are possible for `numpy` arrays, see `pydoc numpy.linalg.norm`).

The file `ft14_nlpoisson_picard.py` contains the complete code for this nonlinear Poisson problem. The implementation is d dimensional, with mesh construction and setting of Dirichlet conditions as explained in Section 6.1.5. For a 33×33 grid with $m = 2$ we need 9 iterations for convergence when the tolerance is 10^{-5}.

3.2.2 A Newton method at the algebraic level

Identifying the variational formulation of the linearized problem. After having discretized our nonlinear PDE problem, we may use Newton's method to solve the system of nonlinear algebraic equations. From the variational problem $F(u;v) = 0 \ \forall v \in \hat{V}$, we get a system of equations for the unknown parameters $U_1, \ldots, U_N$ (by inserting $u = \sum_{j=1}^N U_j \phi_j$ and $v = \hat{\phi}_i$ in $F(u;v)$):

$$F_i(U_1, \ldots, U_N) \equiv \sum_{j=1}^N \int_\Omega \left(q \left(\sum_{\ell=1}^N U_\ell \phi_\ell \right) \nabla \phi_j U_j \right) \cdot \nabla \hat{\phi}_i \, \mathrm{d}x = 0, \quad i = 1, \ldots, N \,. \tag{3.9}$$

Newton's method for the system $F_i(U_1, \ldots, U_j) = 0$, $i = 1, \ldots, N$ can be formulated as

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k, \ldots, U_N^k) \delta U_j = -F_i(U_1^k, \ldots, U_N^k), \quad i = 1, \ldots, N, \tag{3.10}$$

$$U_j^{k+1} = U_j^k + \omega \delta U_j, \quad j = 1, \ldots, N, \tag{3.11}$$

where $\omega \in [0,1]$ is a relaxation parameter, and k is an iteration index. An initial guess u^0 must be provided to start the algorithm.

The original Newton method has $\omega = 1$, but in problems where it is difficult to obtain convergence, so-called *under-relaxation* with $\omega < 1$ may help. It means that one takes a smaller step than what is suggested by Newton's method.

We need, in a program, to compute the Jacobian matrix $\partial F_i/\partial U_j$ and the right-hand side vector $-F_i$. Our present problem has F_i given by (3.9). The derivative $\partial F_i/\partial U_j$ becomes

$$\int_\Omega \left[q'(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^N U_j^k \phi_j)\cdot\nabla\hat{\phi}_i + q\left(\sum_{\ell=1}^N U_\ell^k\phi_\ell\right)\nabla\phi_j\cdot\nabla\hat{\phi}_i\right] \mathrm{d}x\,. \tag{3.12}$$

The following results were used to obtain (3.12):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j}\sum_{j=1}^N U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j}q(u) = q'(u)\phi_j\,. \tag{3.13}$$

We can reformulate the Jacobian matrix in (3.12) by introducing the short notation $u^k = \sum_{j=1}^N U_j^k\phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[q'(u^k)\phi_j\nabla u^k\cdot\nabla\hat{\phi}_i + q(u^k)\nabla\phi_j\cdot\nabla\hat{\phi}_i\right]\mathrm{d}x\,. \tag{3.14}$$

Now it is time to write such a key expression with the computer code friendly notation u^- for u^k:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[q'(u^-)\phi_j\nabla u^-\cdot\nabla\hat{\phi}_i + q(u^-)\nabla\phi_j\cdot\nabla\hat{\phi}_i\right]\mathrm{d}x\,. \tag{3.15}$$

In order to make FEniCS compute this matrix, we need to formulate a corresponding variational problem. Looking at the linear system of equations in Newton's method,

$$\sum_{j=1}^N \frac{\partial F_i}{\partial U_j}\delta U_j = -F_i, \quad i = 1,\ldots,N,$$

we can introduce v as a general test function replacing $\hat{\phi}_i$, and we can identify the unknown $\delta u = \sum_{j=1}^N \delta U_j\phi_j$. From the linear system we can now go "backwards" to construct the corresponding linear discrete weak form $F(u;v) = 0$ to be solved in each Newton iteration:

$$F = \int_\Omega \left[q'(u^-)\delta u\nabla u^-\cdot\nabla v + q(u^-)\nabla\delta u\cdot\nabla v + q(u^-)\nabla u^-\cdot\nabla v\right]\mathrm{d}x\,. \tag{3.16}$$

Note the important feature in Newton's method that the previous solution u^- replaces u in the formulas when computing the matrix $\partial F_i/\partial U_j$ and vector F_i for the linear system in each Newton iteration.

FEniCS implementation. To obtain a good initial guess u^0, we can solve a simplified, linear problem, typically with $q(u) = 1$, which yields the standard Laplace equation $\nabla^2 u^0 = 0$. The code for computing the initial guess u^0 becomes as follows:

```
tol = 1E-14
def left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < tol

def right_boundary(x, on_boundary):
    return on_boundary and abs(x[0]-1) < tol

Gamma_0 = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1 = DirichletBC(V, Constant(1.0), right_boundary)
bcs = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, i.e., m=0)
u = TrialFunction(V)
v = TestFunction(V)
F = dot(grad(u), grad(v))*dx + Constant(0)*v*dx
A, b = assemble_system(lhs(F), rhs(F), bcs)
u_ = Function(V)
U_ = u_.vector()
solve(A, U_, b)
```

Here, `u_` is the initial guess we denote by u^0 in the mathematics.

The Dirichlet boundary conditions for δu, in the problem to be solved in each Newton iteration, are somewhat different than the conditions for u. Assuming that u^k fulfills the Dirichlet conditions for u, δu must be zero at the boundaries where the Dirichlet conditions apply, in order for $u^{k+1} = u^k + \omega\delta u$ to fulfill the right boundary values. We therefore define an additional list of Dirichlet boundary conditions objects for δu:

```
Gamma_0_du = DirichletBC(V, Constant(0), left_boundary)
Gamma_1_du = DirichletBC(V, Constant(0), right_boundary)
bcs_du = [Gamma_0_du, Gamma_1_du]
```

The nonlinear coefficient and its derivative must be defined before coding the weak form of the Newton system:

```
def q(u):
    return (1+u)**m

def Dq(u):
    return m*(1+u)**(m-1)

# Define variational problem in a Newton iteration
du = TrialFunction(V) # u = u_ + omega*du
F = dot(q(u_)*grad(du), grad(v))*dx + \
    dot(Dq(u_)*du*grad(u_), grad(v))*dx + \
    dot(q(u_)*grad(u_), grad(v))*dx

# Newton iteration at the algebraic level
```

The Newton iteration loop is very similar to the Picard iteration loop in Section 3.2.1:

```
du = Function(V)
u  = Function(V)  # u = u_ + omega*du
omega = 1.0       # relaxation parameter
eps = 1.0
tol = 1.0E-5
num_iter = 0
max_iter = 25
# u_ must have right boundary conditions
while eps > tol and iter < max_iter:
    num_iter += 1
    print(num_iter, 'iteration', end=' ')
    A, b = assemble_system(lhs(F), rhs(F), bcs_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print('Norm:', eps)
    u.vector()[:] = u_.vector() + omega*du.vector()
    u_.assign(u)
```

There are other ways of implementing the update of the solution as well:

```
u.assign(u_)  # u = u_
u.vector().axpy(omega, du.vector())

# or
u.vector()[:] += omega*du.vector()
```

The `axpy(a, y)` operation adds a scalar `a` times a `Vector y` to a `Vector` object. It is usually a fast operation calling up an optimized BLAS routine for the calculation.

The important message is that we cannot do (the slightly more intuitive) `u = u_ + omega*du` because this is computed as an UFL expression of a `Function` plus a scalar times a `Function`. We would need to project or interpolate this UFL expression `u` in V in that case. Instead, we update the values through the degrees of freedom vectors of the finite element function objects.

Mesh construction for a d-dimensional problem with arbitrary degree of the Lagrange elements can be done as explained in Section 6.1.5 in [2]. The complete program appears in the file `ft15_nlpoisson_alg_newton.py`.

3.2.3 A Newton method at the PDE level

Although Newton's method in PDE problems is normally formulated at the linear algebra level, i.e., as a solution method for systems of nonlinear algebraic equations, we can also formulate the method at the PDE level. This approach yields a linearization of the PDEs before they are discretized. FEniCS users will probably find this technique simpler to apply than the more

standard method in Section 3.2.2, which needs an unfamiliar interpretation of a linear system as a variational formulation.

The mathematical method. Given an approximation to the solution field, u^k, we seek a perturbation δu so that

$$u^{k+1} = u^k + \delta u \tag{3.17}$$

fulfills the nonlinear PDE. However, the problem for δu is still nonlinear and nothing is gained. The idea is therefore to assume that δu is sufficiently small so that we can linearize the problem with respect to δu. Inserting u^{k+1} in the PDE, linearizing the q term as

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{3.18}$$

and dropping nonlinear terms in δu, we get

$$\nabla\cdot\left(q(u^k)\nabla u^k\right) + \nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = 0\,.$$

We may collect the terms with the unknown δu on the left-hand side,

$$\nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = -\nabla\cdot\left(q(u^k)\nabla u^k\right)\,. \tag{3.19}$$

We may state this equation using the code-friendly notation u^- for u^k:

$$\nabla\cdot\left(q(u^-)\nabla\delta u\right) + \nabla\cdot\left(q'(u^-)\delta u\nabla u^-\right) = -\nabla\cdot\left(q(u^-)\nabla u^-\right)\,. \tag{3.20}$$

The weak form of this PDE is derived by multiplying by a test function v and integrating over Ω, integrating as usual the second-order derivatives by parts. This results in the variational formulation: find $u \in V$ such that $F(\delta u; v) = 0\ \forall v \in \hat{V}$, where

$$F = \int_\Omega \left(q(u^-)\nabla\delta u\cdot\nabla v + q'(u^-)\delta u\nabla u^-\cdot\nabla v\right)\mathrm{d}x - \int_\Omega q(u^-)\nabla u^-\cdot\nabla v\,\mathrm{d}x\,. \tag{3.21}$$

Note that this $F(\delta u; v)$ expression is linear in the unknown δu. We know that FEniCS can neatly extract the corresponding bilinear form and the linear form by the `lhs` and `rhs` commands, so in a program we prefer to state F rather than the bilinear and linear forms.

The function spaces V and $\hat{V}$, being continuous or discrete, are as in the corresponding linear Poisson problem, see Section 2.1.1 in [2].

We must provide some initial guess. In the present problem, the solution of the corresponding linear PDE is a natural choice. The linear PDE corresponds to $q(u) = 1$. The corresponding weak form $a_0(u^0, v) = L_0(v)$ has

$$a_0(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \quad L_0(v) = 0\,.$$

Thereafter, we enter a loop and solve the linearized $F(\delta u; v) = 0$ for δu and compute a new approximation $u^- + \delta u$. Note that δu is a correction, so if the initial guess u^0 satisfies the prescribed Dirichlet conditions on some part Γ_D of the boundary, we must demand $\delta u = 0$ on Γ_D.

FEniCS implementation. Looking at (3.21) we realize that the variational form is the same as for the Newton method at the algebraic level in Section 3.2.2. Since Newton's method at the algebraic level required some "backward" construction of the underlying weak forms, FEniCS users may prefer Newton's method at the PDE level, which this author finds more straightforward, although not so commonly documented in the literature on numerical methods for PDEs. There is seemingly no need for differentiations to derive a Jacobian matrix, but a mathematically equivalent derivation is done when nonlinear terms are linearized using the first two Taylor series terms and when products in the perturbation δu are neglected.

The implementation is identical to the one in Section 3.2.2 and is found in the file `ft16_nlpoisson_pde_newton.py`. The reader is encouraged to go through this code to be convinced that the present method actually ends up with the same program (`ft15_nlpoisson_alg_newton.py`) as needed for the Newton method at the linear algebra level in Section 3.2.2.

3.2.4 Implementations with functions and classes

The implementations of the Picard and Newton methods so far in this chapter have been done in terms of flat programs to maximize the focus on the numerical details. However, for real-life applications of the software you will most likely prefer to have a solver function or problem and solver classes.

The Newton methods implemented as solver functions are found in the file `ft13_nlpoisson_func.py`. The functions `solver`, `application_test` and `test_solver` should be examined. These represent a general solver, a special application, and a series of unit tests, respectively.

A downside of the solver function mentioned is that it is quite restricted with respect to boundary conditions and meshes. Much more flexibility is enabled through a class implementation, although the code is also much more lengthy. The file `ft13_nlpoisson_func.py` contains all the details:

- class `NonlinearPoissonSolver` for solving $-\nabla\cdot(q(u)\nabla u) = f$ on a general mesh with general Dirichlet and Neumann conditions,
- class `NonlinearPoissonProblem` as an abstract super class for nonlinear Poisson problems,

- class `TestProblem` for a specific implementation of the test problem used in this chapter.

There is also a comprehensive unit test for verifying all methods through empirical convergence rate estimation. We strongly recommend the reader to study the class implementation to identify its superiority to other implementations in this chapter and to use this code base as ideas for solving nonlinear problems with FEniCS.

Chapter 4
Mixed finite element programming

Chapter 5
High-performance computing

How to take an ordinary FEniCS program, profile it, optimize the code, and turn into an HPC application for parallel platforms.

Chapter 6
Multi-physics in multi-domains

Systems of PDEs, coupling of domains, typically elasticity with a larger domain for heat transfer, simple fluid-structure interaction?

Chapter 7
More old stuff

7.1 Solving Neumann problems

Could have a section here with pure Neumann problems and different methods for handling that case:

- Fixing the value at a point.
- Integral constraint: vanishing average value of u for instance.
- Others.

Applications: Potential theory in fluid flow. Always pure Neumann problems and (most likely) a headache for teachers how to code demos.

7.2 Transforming hypercube domains

The uniform hypercube domains produced by functions like `UnitIntervalMesh`, `UnitSquareMesh`, `UnitCubeMesh`, `IntervalMesh`, `RectangleMesh`, and `BoxMesh` can with simple mathematical means be transformed to other shapes, or coordinates can be stretched to provide a better resolution.

7.2.1 Coordinate stretching

A mesh that is denser toward a boundary is often desired to increase accuracy in that region. Given a mesh with uniformly spaced coordinates $x_0, \ldots, x_{M-1}$ in $[a,b]$, the coordinate transformation $\xi = (x-a)/(b-a)$ maps x onto $\xi \in [0,1]$. A new mapping $\eta = \xi^s$, for some $s > 1$, stretches the mesh toward $\xi = 0$ ($x = a$), while $\eta = \xi^{1/s}$ makes a stretching toward $\xi = 1$ ($x = b$). Mapping the $\eta \in [0,1]$ coordinates back to $[a,b]$ gives new, stretched x coordinates,

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{s} \tag{7.1}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{7.2}$$

toward $x = b$. Figure 7.1 shows the effect of making a rectangular mesh denser toward $x = 0$ (prior to the coordinate transformation below).

7.2.2 Rectangle to hollow circle mapping

One way of creating more complex geometries is to transform the vertex coordinates in a rectangular mesh according to some formula. Say we want to create a part of a hollow cylinder of Θ degrees, with inner radius a and outer radius b. A standard mapping from polar coordinates to Cartesian coordinates can be used to generate the hollow cylinder. Given a rectangle in $(\bar{x}, \bar{y})$ space such that $a \leq \bar{x} \leq b$ and $0 \leq \bar{y} \leq 1$, the mapping

$$\hat{x} = \bar{x}\cos(\Theta\bar{y}), \quad \hat{y} = \bar{x}\sin(\Theta\bar{y}),$$

takes a point in the rectangular $(\bar{x}, \bar{y})$ geometry and maps it to a point $(\hat{x}, \hat{y})$ in a hollow cylinder.

The corresponding Python code for first stretching the mesh and then mapping it onto a hollow cylinder looks as follows:

```
Theta = pi/2
a, b = 1, 5.0
nr = 10  # divisions in r direction
nt = 20  # divisions in theta direction
mesh = RectangleMesh(a, 0, b, 1, nr, nt, 'crossed')

# First make a denser mesh towards r=a
x = mesh.coordinates()[:,0]
y = mesh.coordinates()[:,1]
s = 1.3

def denser(x, y):
    return [a + (b-a)*((x-a)/(b-a))**s, y]

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
plot(mesh, title='stretched mesh')

def cylinder(r, s):
```

```
    return [r*numpy.cos(Theta*s), r*numpy.sin(Theta*s)]

x_hat, y_hat = cylinder(x_bar, y_bar)
xy_hat_coor = numpy.array([x_hat, y_hat]).transpose()
mesh.coordinates()[:] = xy_hat_coor
plot(mesh, title='hollow cylinder')
interactive()
```

The result of calling `denser` and `cylinder` above is a list of two vectors, with the x and y coordinates, respectively. Turning this list into a `numpy` array object results in a $2 \times M$ array, M being the number of vertices in the mesh. However, `mesh.coordinates()` is by a convention an $M \times 2$ array so we need to take the transpose. The resulting mesh is displayed in Figure 7.1.

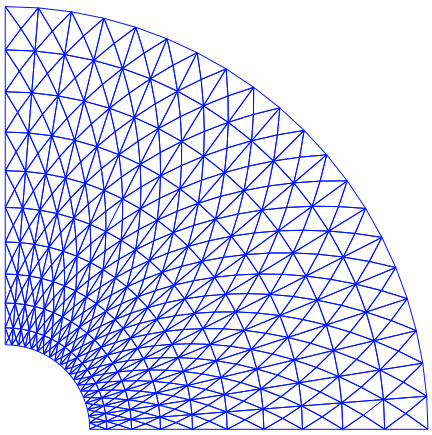

Fig. 7.1 Hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

Setting boundary conditions in meshes created from mappings like the one illustrated above is most conveniently done by using a mesh function to mark parts of the boundary. The marking is easiest to perform before the mesh is mapped since one can then conceptually work with the sides in a pure rectangle.

7.3 Encyclopedia stuff

7.3.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenicsproject.org`). Some components are DOLFIN and Viper, explicitly referred to in this tutorial. Others are FFC and FIAT, heavily used by the programs appearing in this tutorial, but never explicitly used from the programs.

DOLFIN: a FEniCS component, more precisely a C++ library, with a Python interface, for performing important actions in finite element programs. DOLFIN makes use of many other FEniCS components and many external software packages.

Viper: a FEniCS component for quick visualization of finite element meshes and solutions.

UFL: a FEniCS component implementing the *unified form language* for specifying finite element forms in FEniCS programs. The definition of the forms, typically called `a` and `L` in this tutorial, must have legal UFL syntax. The same applies to the definition of functionals (see Section 6.3.4).

Class (Python): a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

Instance (Python): an object of a particular type, where the type is implemented as a class. For instance, `mesh = UnitIntervalMesh(10)` creates an instance of class `UnitIntervalMesh`, which is reached by the name `mesh`. (Class `UnitIntervalMesh` is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

Class method (Python): a function in a class, reached by dot notation: `instance_name.method_name`

argument `self` (Python): required first parameter in class methods, representing a particular object of the class. Used in method definitions, but never in calls to a method. For example, if `method(self, x)` is the definition of `method` in a class `Y`, `method` is called as `y.method(x)`, where `y` is an instance of class `Y`. In a call like `y.method(x)`, `method` is invoked with `self=y`.

Class attribute (Python): a variable in a class, reached by dot notation: `instance_name.attribute_name`

7.3.2 Overview of objects and functions

Most classes in FEniCS have an explanation of the purpose and usage that can be seen by using the general documentation command `pydoc` for Python objects. You can type

Terminal

```
pydoc fenics.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquareMesh`, `Function`, `Viper`, etc.). Below is an overview of the most important classes and functions in FEniCS programs, in the order they typically appear within programs.

`UnitSquareMesh(nx, ny)`: generate mesh over the unit square $[0,1] \times [0,1]$ using `nx` divisions in x direction and `ny` divisions in y direction. Each of the `nx*ny` squares are divided into two cells of triangular shape.

`UnitIntervalMesh`, `UnitCubeMesh`, `UnitCircleMesh`, `UnitSphere`, `IntervalMesh`, `RectangleMesh`, and `BoxMesh`: generate mesh over domains of simple geometric shape, see Section 7.2.

`FunctionSpace(mesh, element_type, degree)`: a function space defined over a mesh, with a given element type (e.g., `'Lagrange'` or `'DG'`), with basis functions as polynomials of a specified degree.

`Expression(formula, p1=v1, p2=v2, ...)`: a scalar- or vector-valued function, given as a mathematical expression `formula` (string) written in C++ syntax. The spatial coordinates in the expression are named `x[0]`, `x[1]`, and `x[2]`, while time and other physical parameters can be represented as symbols `p1`, `p2`, etc., with corresponding values `v1`, `v2`, etc., initialized through keyword arguments. These parameters become attributes, whose values can be modified when desired.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`. If `V` is a `FunctionSpace` object, `Function(V)` becomes a scalar field, and with `V` as a `VectorFunctionSpace` object, `Function(V)` becomes a vector field.

`SubDomain`: class for defining a subdomain, either a part of the boundary, an internal boundary, or a part of the domain. The programmer must subclass `SubDomain` and implement the `inside(self, x, on_boundary)` function (see Section 2.2) for telling whether a point `x` is inside the subdomain or not.

`Mesh`: class for representing a finite element mesh, consisting of cells, vertices, and optionally faces, edges, and facets.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Section 4.2) or for boundary conditions (see Section 4.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing

the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

`TestFunction(V)`: define a test function on a space `V` to be used in a variational form.

`TrialFunction(V)`: define a trial function on a space `V` to be used in a variational form to represent the unknown in a finite element problem.

`assemble(X)`: assemble a matrix, a right-hand side, or a functional, given a from `X` written with UFL syntax.

`assemble_system(a, L, bcs)`: assemble the matrix and the right-hand side from a bilinear (`a`) and linear (`L`) form written with UFL syntax. The `bcs` parameter holds one or more `DirichletBC` objects.

`LinearVariationalProblem(a, L, u, bcs)`: define a variational problem, given a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` objects stored in `bcs`.

`LinearVariationalSolver(problem)`: create solver object for a linear variational problem object (`problem`).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` object), `U` as unknown (`Vector` object), and `b` as right-hand side (`Vector` object). Usually, `U = u.vector()`, where `u` is a `Function` object representing the unknown finite element function of the problem, while `A` and `b` are computed by calls to `assemble` or `assemble_system`.

`plot(q)`: quick visualization of a mesh, function, or mesh function `q`, using a built-in, VTK-based component in FEniCS.

`interpolate(func, V)`: interpolate a formula or finite element function `func` onto the function space `V`.

`project(func, V)`: project a formula or finite element function `func` onto the function space `V`.

7.3.3 Handy methods in key FEniCS objects

In general, `pydoc fenics.X` shows the documentation of any name `X` and lists all the methods (i.e.g, functions in the class) that can be called. Below, we list just a few, but very useful, methods in the most central FEniCS classes.

Mesh. Let `mesh` be a `Mesh` object.

- `mesh.coordinates()` returns an array of the coordinates of the vertices in the mesh.
- `mesh.num_cells()` returns the number of cells (triangles) in the mesh,
- `mesh.num_vertices()` returns the number of vertices in the mesh (with our choice of linear Lagrange elements this equals the number of nodes, `len(u_array)`, or dimension of the space `V.dim()`),

- `mesh.cells()` returns the vertex numbers of the vertices in each cell as a `numpy` array with shape (*number of cells*, *number of vertices in a cell*),
- `mesh.hmin()` returns the minimum cell diameter ("smallest cell"),
- `mesh.hmax()` returns the maximum cell diameter ("largest cell").
- `mesh.topology().dim()` returns the number of physical dimensions of the mesh.

Writing `print(mesh)` dumps a short, pretty-print description of the mesh (`print(mesh)` actually displays the result of `str(mesh)`, which defines the pretty print):

```
<Mesh of topological dimension 2 (triangles) with
16 vertices and 18 cells, ordered>
```

Function space. Let `V` be a `FunctionSpace` object.

- `V.mesh()` returns the associated mesh.
- `V.dim()` returns the dimension (number of degrees of freedom).
- `V.ufl_element()` returns the associated finite element.

Function. Let `u` be a `Function` object.

- `u.function_space()` returns the associated function space.
- `u.vector()` returns the vector object of degrees of freedom.
- `u.vector().array()` returns a copy of the degrees of freedom in a `numpy` array.

7.3.4 Using a backend-specific solver

Warning

The linear algebra backends in FEniCS have recently changed. This section is not yet up-to-date.

The linear algebra backend determines the specific data structures that are used in the `Matrix` and `Vector` classes. For example, with the PETSc backend, `Matrix` encapsulates a PETSc matrix storage structure, and `Vector` encapsulates a PETSc vector storage structure. Sometimes one wants to perform operations directly on (say) the underlying PETSc objects. These can be fetched by

```
A_PETSc =
down_cast(A).mat() b_PETSc = down_cast(b).vec() U_PETSc =
down_cast(u.vector()).vec()
```

Here, `u` is a `Function`, `A` is a `Matrix`, and `b` is a `Vector`. The same syntax applies if we want to fetch the underlying Epetra, uBLAS, or MTL4 matrices and vectors.

Sometimes one wants to implement tailored solution algorithms, using special features of the underlying numerical packages. Here is an example where we create an ML preconditioned Conjugate Gradient solver by programming with Trilinos-specific objects directly. Given a linear system $AU = b$, represented by a `Matrix` object `A`, and two `Vector` objects `U` and `b` in a Python program, the purpose is to set up a solver using the Aztec Conjugate Gradient method from Trilinos' Aztec library and combine that solver with the algebraic multigrid preconditioner ML from the ML library in Trilinos. Since the various parts of Trilinos are mirrored in Python through the PyTrilinos package, we can operate directly on Trilinos-specific objects.

```
try:
    from PyTrilinos import Epetra, AztecOO, TriUtils, ML
except:
    print('''You Need to have PyTrilinos with
Epetra, AztecOO, TriUtils and ML installed
for this demo to run''')
    exit()

from fenics import *

if not has_la_backend('Epetra'):
    print('Warning: Dolfin is not compiled with Trilinos')
    exit()

parameters['linear_algebra_backend'] = 'Epetra'

# create matrix A and vector b in the usual way
# u is a Function

# Fetch underlying Epetra objects
A_epetra = down_cast(A).mat()
b_epetra = down_cast(b).vec()
U_epetra = down_cast(u.vector()).vec()

# Sets up the parameters for ML using a python dictionary
ML_param = {"max levels"        : 3,
            "output"            : 10,
            "smoother: type"    : "ML symmetric Gauss-Seidel",
            "aggregation: type" : "Uncoupled",
            "ML validate parameter list" : False
}

# Create the preconditioner
prec = ML.MultiLevelPreconditioner(A_epetra, False)
prec.SetParameterList(ML_param)
prec.ComputePreconditioner()

# Create solver and solve system
```

```
solver = AztecOO.AztecOO(A_epetra, U_epetra, b_epetra)
solver.SetPrecOperator(prec)
solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
solver.SetAztecOption(AztecOO.AZ_output, 16)
solver.Iterate(MaxIters=1550, Tolerance=1e-5)

plot(u)
```

Chapter 8
Troubleshooting

hpl 10: We don't have pages for this, but the document can appear as additional material on the web.

8.1 Compilation Problems

Expressions and variational forms in a FEniCS program need to be compiled to C++ and linked with libraries if the expressions or forms have been modified since last time they were compiled. The tool Instant, which is part of the FEniCS software suite, is used for compiling and linking C++ code so that it can be used with Python.

Sometimes the compilation fails. You can see from the series of error messages which statement in the Python program that led to a compilation problem. Make sure to scroll back and identify whether the problematic line is associated with an expression, variational form, or the solve step.

The final line in the output of error messages points to a log file from the compilation where one can examine the error messages from the compiler. It is usually the last lines of this log file that are of interest. Occasionally, the compiler's message can quickly lead to an understanding of the problem. A more fruitful approach is normally to examine the below list of common compilation problems and their remedies.

8.1.1 Problems with the Instant cache

Instant remembers information about previous compilations and versions of your program. Sometimes removal of this information can solve the problem. Just run

Terminal

in a terminal window whenever you encounter a compilation problem.

8.1.2 Syntax errors in expressions

If the compilation problem arises from line with an `Expression` object, examine the syntax of the expression carefully. Section 2.3 contains some information on valid syntax. You may also want to examine the log file, pointed to in the last line in the output of error messages. The compiler's message about the syntax problem may lead you to a solution.

Some common problems are

1. using `a**b` for exponentiation (illegal in C++) instead of `pow(a, b)`,
2. forgetting that the spatial coordinates are denoted by a vector `x`,
3. forgetting that the x, y, and z coordinates in space correspond to `x[0]`, `x[1]`, and `x[2]`, respectively.

Failure to initialize parameters in the expressions lead to a compilation error where this problem is explicitly pointed out.

Example. The implementation

```
u_exact = Expression(
    'x[1] <= 0.5? 2*x[1]*p_1/(p_0 + p_1) : '
    '((2*x[1]-1)p_0 + p_1)/(p_0 + p_1)',
    p_0=p_values[0], p_1=p_values[1])
```

fails with compilation error

```
RuntimeError: In instant.recompile: The module did not compile with
command 'make VERBOSE=1', see '/some/path/.../compile.log'
```

Looking up the `compile.log` file and searching for `error`, we see the following message from the C++ compiler:

```
error: expected ‘’) before ‘’p_0
   values[0] = x[1] <= 0.5? 2*x[1]*p_1/(p_0 + p_1) :
   ((2*x[1]-1)p_0 + p_1)/(p_0 + p_1);
                ^
```

Now we realize that a `*` symbol is missing between `)` and `p_0`.

8.1.3 An integral without a domain is now illegal

This error message from UFL (`ufl.log.UFLException`) often points to a factor in a variational form that is 0. The factor should be `Constant(0)`.

8.1.4 Problems with SymPy and `diff` expressions

If you work with FEniCS and SymPy in the same program, be aware that `from fenics import *` imports a lot of symbols that you also have in `sympy`, so you have to do `import sympy as sym`. However, it is easy to write something like `p*diff(u,x)` meaning differentiation of the symbolic expression `u`, but a pure `diff` is then the `diff` object from `fenics`, not from `sympy`. The right statement is `p*sym.diff(u,x)`. (Error messages arising from this example can be quite difficult to interpret).

8.1.5 Problems in the solve step

Sometimes the problem lies in the solve step where a variational form is turned into a system of algebraic equations. The error message *Unable to extract all indicies* points to a problem with the variational form. Common errors include

1. missing either the `TrialFunction` or the `TestFunction` object,
2. no terms without `TrialFunction` objects.
3. mathematically invalid operations in the variational form.

The first problem implies that one cannot make a matrix system or system of nonlinear algebraic equations out of the variational form. The second problem means that there is no "right-hand side" terms in the PDE with known quantities. Sometimes this is seemingly the case mathematically because the "right-hand side" is zero. Variational forms must represent this case as `Constant(0)*v*dx` where `v` is a `TestFunction` object. An example of the third problem is to take the `inner` product of a scalar and a vector (causing in this particular case the error message to be "Shape mismatch").

The message *Unable to extract common cell; missing cell definition in form or expression* will typically arise from a term in the form where a test function (holding mesh and cell information) is missing. For example, a zero right-hand side `Constant(0)*dx` will generate this error.

8.1.6 Unable to convert object to a UFL form

One common reason for the above error message is that a form is written without being multiplied by `dx` or `ds`.

8.1.7 UFL reports that a numpy array cannot be converted to any UFL type

One reason may be that there are mathematical functions like `sin` and `exp` operating on `numpy` arrays. The problem is that the

```
from fenics import *
```

statement imports `sin`, `cos`, and similar mathematical functions from UFL and these are aimed at taking `Function` or `TrialFunction` objects as arguments and not `numpy` arrays. The remedy is to use prefix mathematical functions aimed at `numpy` arrays with `numpy`, or `np` if `numpy` is imported as `np`: `numpy.exp` or `np.exp`, for instance. Normally, boundary conditions and analytical solutions are represented by `Expression` objects and then this problem does not arise. The problem usually arises when pure Python functions with, e.g., analytical solutions are introduced for, e.g., plotting.

8.1.8 All programs fail to compile

When encountering a compilation problem where the Instant log file says something about missing double quote in an `Expression`, try compiling a previously working program. If that program faces the same problem, reboot Ubuntu and try again. If the problem persists, try running the Update Manager (because unfinished updates can cause compiler problems), reboot and try again.

8.2 Problems with Expression Objects

8.2.1 There seems to be some bug in an Expression object

Run the command `instant-clean` to ensure that everything is (re)compiled. Check the formulas in string expressions carefully, and make sure that divisions do not lead to integer division (i.e., at least one of the operands in a division must be a floating-point variable).

8.2.2 Segmentation fault when using an Expression object

One reason may be that the point vector `x` has indices out of bounds, e.g., that you access `x[2]` but the mesh is only a 2D mesh. Also recall that the components of `x` are `x[0]`, `x[1]`, etc. Accessing `x[2]` as the "y" coordinate is a common error.

8.3 Other Problems

8.3.1 Very strange error message involving a `mesh` variable

If you encounter a really strange error message, and the statement in question involves a variable with name `mesh`, check if this is really your mesh variable. When doing `from fenics import *`, you get a `mesh` variable, which is actually a module, and sending this module to functions creates a `TypeError`. Substitute with the actual name of your mesh object.

One should also note other names that get imported by `from fenics import *`: `e`, `f`, `i`, `j`, `k`, `l`, `p`, `q`, `r`, `s`. It is easy to use such variables without initializing them, and strange error message arises (since the mentioned names are UFL `Index` objects).

8.3.2 The plot disappears quickly from the screen

You have forgotten to insert `interactive()` as the last statement in the program.

8.3.3 Only parts of the program are executed

Check if a call to `interactive()` appears in the middle of the program. The computations are halted by this call and not continued before you press `q` in a plot window. Most people thus prefer to have `interactive()` as the last statement.

8.3.4 Error in the definition of the boundary

Consider this code and error message:

```
class DirichletBoundary(SubDomain):  # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary and abs(x) < 1E-14

bc = DirichletBC(V, u0, xleft_boundary)

Error: ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

The reason for this error message is that `x` is a point vector, not just a number. In the `inside` function one must work with the components of `x`: `x[0]`, `x[1]`, etc.

8.3.5 The solver in a nonlinear problems does not converge

There can be many reasons for this common problem:

1. The form (variational formulation) is not consistent with the PDE(s).
2. The boundary conditions in a Newton method are wrong. The correction vector must have vanishing essential conditions where the complete solution has zero or non-zero values.
3. The initial guess for the solution is not appropriate. In some problems, a simple function equal to 0 just leads to a zero solution or a divergent solver. Try 1 as initial guess, or (better) try to identify a linear problem that can be used to compute an appropriate initial guess, see Section 3.2.2.

8.4 How To Debug a FEniCS Program?

Here is an action list you may follow.

Step 1. Examine the weak form and its implementation carefully. Check that all terms are multiplied by `dx` or `ds`, and that the terms do not vanish; check that at least one term has both a `TrialFunction` and a `TestFunction` (term with unknown); and check that at least one term has no `TrialFunction` (known term).

Step 2. Check that Dirichlet boundary conditions are set correctly.

```
# bcs is list of DirichletBC objects
for bc in bcs:
bc_dict = bc.get_boundary_values()
    for dof in bc_dict:
        print('dof %d: value=%s' % (dof, bc_dict[dof]))
```

See also an expanded version of this snippet in the `solvers_bc` `ft08_poisson_vc.py` file located in the directory `poisson`.

A next step in the debugging, if these values are wrong, is to call the functions that define the boundary parts. For example,

```
for coor in mesh.coordinates():
    if my_boundary_function(coor, True):
        print('%s is on the boundary' % coor)

# or, in case of a SubDomain subclass my_subdomain_object,
for coor in mesh.coordinates():
    if my_subdomain_object.inside(coor, True):
        print('%s is on the boundary' % coor)
```

You may compute points along a line and plot the solution along this line (see the `prog['membrane'].py` example). You may also print out the Dirichlet values as shown in the `prog['heat'].py` and `prog['poisson_vc'].py` files.

Step 4. Switching to a simple set of coefficients and boundary conditions, such that the solution becomes simple too, but still obeys the same PDE, may help since it is then easier to examine numerical values in the solution array.

Step 5. Formulate a corresponding 1D problem. Often this can be done by just running the problem with a 1D mesh. Doing hand calculations of element matrices and vectors, and comparing the assembled system from these hand calculations with the assembled system from the FEniCS program can uncover bugs. For nonlinear problems, or problems with variable coefficients, it is usually wise to choose simple coefficients so that the problem becomes effectively linear and the hand calculations are doable.

8.5 To-do list

8.5.1 AL list

- Title?: Writing State-of-the-Art Finite Element Solvers in Minutes (HPL: Yes - let's have this for a while)
- Title: "Writing... in minutes" sounds more like a good subtitle than an actual title. We already have a good subtitle in "The FEniCS tutorial" so let's try to find a more "serious" main title. (HPL: What about switching?)
- Title?: Introduction to Finite Element Programming

- Add 2nd author
- Rewrite abstract to reflect new title and non-initial state of book
- Rename and expand Chapter 5: Boundary conditions, markers and subdomains (many and confusing ways to do this, `MeshFunctions`, `MeshData`, etc) (HPL: This is important and confusing, but very much needed early on)
- DONE: Add chapter on installation (difficult, still in flux)
- Installation chapter should be number 0
- Remove capitalization of sections in Chapter 8
- Add new chapter: "Advanced example: Linear elasticity" (HPL: do this first - important for many courses)
- Add new chapter: "Advanced example: Hyperelasticity"
- Add new chapter: "Advanced example: Incompressible Navier-Stokes"
- Place these chapters near end of book so progression becomes:
  - Installation
  - Fundamentals (3 chapters: Fundamentals, Time-dependent, Nonlinear)
  - Misc technical chapters (boundary conditions, geometries etc) (HPL: Think we need these things before time-dependent, or maybe split the very simplest things from time-dependent (1st example) and nonlinear (`solve(F==0, ...)`) and place in fundamentals?)
  - Advanced examples (using previously discussed techniques)
- Add chapter on performance and profiling? HPL: Yes!
- Move source code links to `http://fenisproject.org/tutorial` (we can set up a cron job to copy files from `hplgit`)
- Number examples sequentially, something like
  - `fenics_tutorial_01_poisson_first.py`
  - `fenics_tutorial_05_poisson_nonlinear.py`

  Hierarchic structure becomes complicated, better with flat structure (HPL: These filenames are too long, giving too long lines in the text and the need for lots of rewrites. We have an automated scheme to assign new filenames to the logical files that we use for the text. Right now the names are `ft01_poisson.py`.)
- Add command `fenics-version`
- DONE: Use dot and grad in Poisson example (not `inner` and `nabla_grad`)
- Use new notation `'P'` instead of `'Lagrange'` in `FunctionSpace`. HPL: Done. (HPL: Not yet documented in `FunctionSpace` ;-)
- DONE: Use `from fenics import *`
- Fix copyright footer so we avoid the linebreak. HPL: Fixed. This and similar adjustments in the latex file is easily done by auto editing in `make.sh`.
- How does doconce handle – and —? HPL: Like latex. Just use ndash if you want. ndash and mdash are ignored in other formats than latex and html where there is no support for this. Just introduce ndash where you like, as in Navier–Stokes.

- When to use . and when to use . ? HPL: Springer had a rule with thinspace. I don't know if it applies to the brief series, but it became a habit of mine since other styles demanded it.
- When to specify floats as 2 or 2.0? HPL: In code? Anytime there is a danger of integer division.
- When do talk about a function `foo` or `foo()`. HPL: If parameters in any way are considered important, include them, otherwise just the function name. (Need to know info.)
- varproblem reads: find ... or varproblem reads: Find ...? HPL: I use lower case after colon, even if full sentence - that's more usual in English I was told than our Scandinavian style.
- Need to include example of mixed problems (systems) in book 1. HPL: Agree.
- Consistency check for mesh size: 16 x 16 good choice overall? HPL: Depends. If we have exact numerical solution, a very coarse mesh suffices, otherwise we should have at least two meshes and report the error fraction. But 16 x 16 sounds reasonable as numbers in the demo programs.
- Add little box in all examples listing what new features are explained for the particular example. HPL: Good idea!
- Should all code files be included in the text? HPL: No, we run out of space. Can do it in the beginning. Later, the codes also grow in size, and I think codes that span two pages or more give a "boring" look in the book. People won't read it... Instead we use the mako construction to link to the repo file. It has nice syntax highlighting, comes up in a separate window if you want, and can be downloaded. Maybe we need to explain "raw" the first time we link to a github file?
- Possible extra appendix chapters (but we don't seem to have enough space): List of finite element spaces + picture of Periodic Table, List of UFL operators, List of variational forms for common PDE, Notation and conventions for simplices, List of most useful FEniCS classes and most useful member functions: Mesh, FunctionSpace, VectorFunctionSpace, DirichletBC, etc.

8.5.2 HPL list

Remember: cannot exceed 150 pages (as reported at the end of `ftut.log`). Solutions to exercises will not appear in the printed book. If we run out of pages, we can also remove the exercises by putting if-else constructs around them. There will be one short printed tutorial and then extended e-versions with exercises (and optionally solution) on our github web site.

Regarding layout, we must use `svmono.cls` for the printed book, but are allowed to use gray background in code boxes and lmodern instead of Courier

for monospace font. Springer's official ebook has exactly the same layout. For all other versions on our github web site, we can choose whatever layout we want.

- `mshr` must align subdomains with cell facets!! Important feature.
- Demonstrate that meshes and mesh functions can be stored in XML files.
- Programs are now flat demos. Educate the reader with better software engineering habits: functions, classes, unit tests. Avoid copy-and-edit flat programs implied by today's collection. (HPL)
- Find successful exercises from various tutorials (AL) and add as exercises in the book (HPL/AL). Exercises are key for learning software so having them (in an extended version?) sends an important signal about their relevance.
- According to the `plot` doc string, it should be easy to plot the element a la `plot(u.function_space().ufl_element())` but I did not get this to work. Not crucial, but plotting the element is a nice feature :-)
- Comment regarding FEniCS demos: The documented demos mention a lot of packages: DOLFIN, FFC, Fiat, ... Make sure the reader of the tutorial does not get lost in the jungle of packages and make sure the names are explained somewhere in the text such that the tutorial is a good background for understanding every demo in every detail.
- Introduce `near(x, x0, eps)`‘ earlier? Not used until it got the `eps` argument, but is more readable. Important to understand the underlying problem with rounding which is more explicit when doing `abs(x-x0) < eps`.
- Can we change the value of `DOLFIN_EPS`? `imort fenics; fenics.DOLFIN_EPS=...` will work, but then all modules in the simulator must do `import fenics`. Note that its value is very strict, e.g., `10.1+10.2` has rounding $3.5 10^{-15}$, so `DOLFIN_EPS` is strictly for scaled problems only, where all variables are in $[0,1]$.

8.5.3 HPL questions

Iterative linear solvers info. We can get this printed out on the screen, but is there any method to extract this text inside the program, such that we can see how many Krylov iterations we do etc.? Any way for Python to capture the standard output stream in C++?

Easy to write a script that post-processes the output, but we have to wait until the simulator has terminated, or we can pipe to a script `process.py` that treats the output in some desired way (could append some into to a file and that is reopened by the simulator):

Terminal

```
Terminal> python mysolver.py | process.py
```

where the simplest `process.py` is

```
import sys, time
t0 = time.clock()
while True:
    line = sys.stdin.readline()
    if not line:
        break
    t1 = time.clock()
    print 'after %g seconds: %s' % (t1-t0, line.rstrip())
print 'END'
```

References

[1] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, fifth edition, 2016.

[2] H. P. Langtangen and A. Logg. *The FEniCS Tutorial - Writing State-of-the-art Finite Element Solvers in Minutes*. Springer, 2016.

[3] H. P. Langtangen and G. K. Pedersen. *Scaling of Differential Equations*. Simula Springer Brief Series. Springer, 2016. `http://tinyurl.com/qfjgxmf/web`.

Index

assembly, increasing efficiency, 17
assert, 4
attribute (class), 86
automatic differentiation, 65

class, 86
compilation problems, 93
coordinate stretching, 83
coordinate transformations, 83

`derivative`, 65
DOLFIN, 86
down-casting matrices and vectors, 89

Epetra, 90

FEniCS, 86
`ft13_nlpoisson_func.py`, 64
`ft14_nlpoisson_picard.py`, 69
`ft15_nlpoisson_alg_newton.py`, 71
`ft16_nlpoisson_pde_newton.py`, 75

Gateaux derivative, 65

instance, 86

Jacobian, automatic computation, 65
Jacobian, manual computation, 70

mesh transformations, 83
method (class), 86

Newton's method (algebraic equations), 70
Newton's method (PDE level), 73
nonlinear variational problems, 64
`NonlinearVariationalProblem`, 64
`NonlinearVariationalSolver`, 64

PETSc, 89
Picard iteration, 68
plotting problems, 97
`pydoc`, 87, 88

`self`, 86
successive substitutions, 68

Trilinos, 90
troubleshooting, 93

UFL, 86
under-relaxation, 70

Viper, 86