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

Writing Advanced State-of-the-Art Finite Element Solvers in Hours - Advanced FEniCS Programming

Apr 19, 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.

Chapter 1
Python programming and PDE solver design

hpl 1: I don't like this title, but have no other good alternative... **AL 2**: Experimenting with new title

1.1 Refactored implementation

Our first programs in this book are all "flat". That is, they are not organized into logical, reusable units in terms of Python functions. Such flat programs are popular for quickly testing out some software, but not well suited for serious problem solving. We shall therefore at once *refactor* the program, meaning that we divide it into functions, but this is just a reordering of the existing statements. During refactoring, we try make functions as reusable as possible in other contexts, but statements specific to a certain problem or task are also encapsulated in (non-reusable) functions. Being able to distinguish reusable code from specialized code is a key issue when refactoring code, and this ability depends on a good mathematical understanding of the problem at hand ("what is general, what is special?"). In a flat program, general and specialized code (and mathematics) is often mixed together.

1.1.1 A general solver function

We consider the flat program developed in Section **??**. Some of the code in this program is needed to solve any Poisson problem $-\nabla^2 u = f$ on $[0,1]\times[0,1]$ with $u = u_0$ on the boundary, while other statements arise from our simple test problem. Let us collect the general, reusable code in a function `solver`. Our special test problem will then just be an application of `solver` with some additional statements. We limit the `solver` function to just *compute the nu-*

merical solution. Plotting and comparing the solution with the exact solution are considered to be problem-specific activities to be performed elsewhere.

We parameterize `solver` by f, u_0, and the resolution of the mesh. Since it is so trivial to use higher-order finite element functions by changing the third argument to `FunctionSpace`, we let also the degree of the polynomials in the finite element basis functions be an argument to `solver`.

```
from fenics import *

def solver(f, u0, Nx, Ny, degree=1):
    """
    Solve -Laplace(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u=u0 (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = dot(grad(u), grad(v))*dx
    L = f*v*dx

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

    return u
```

Plotting for the test problem. The additional tasks we did in our initial program can be placed in other functions. For example, plotting the solution in our particular test problem is placed in an `application_test` function:

```
def application_test():
    """Plot the solution in the test problem."""
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    u = solver(f, u0, 6, 4, 1)
    # Dump solution to file in VTK format
    u.rename('u', 'u')  # name 'u' is used in plot
    vtkfile = File("poisson.pvd")
    vtkfile << u
    # Plot solution and mesh
    plot(u)
```

Make a module! The refactored code is put in a file `ft04_poisson_func.py`. We should make sure that such a file can be imported (and hence reused) in other programs. Then all statements in the main program should appear with a test `if __name__ == '__main__':`. This test is true if the file is executed as a program, but false if the file is imported. If we want to run this file in the same way as we can run `ft04_poisson_func.py`, the main program is simply a call to `application_test()` followed by a call `interactive()` to hold the plot:

```
if __name__ == '__main__':
    application_test()
    # Hold plot
    interactive()
```

1.1.2 Verification and unit tests

The remaining part of our first program is to compare the numerical and the exact solution. Every time we edit the code we must rerun the test and examine that `max_error` is sufficiently small so we know that the code still works. To this end, we shall adopt *unit testing*, meaning that we create a mathematical test and corresponding software that can run all our tests automatically and check that all tests pass. Python has several tools for unit testing. Two very popular ones are pytest and nose. These are almost identical and very easy to use. More classical unit testing with test classes is offered by the built-in tool `unittest`, but here we are going to use pytest (or nose) since it demands shorter and clearer code.

Mathematically, our unit test is that the finite element solution of our problem when $f = -6$ equals the exact solution $u = u_0 = 1 + x^2 + 2y^2$. We have already created code that finds the maximum error in the numerical solution. Because of rounding errors, we cannot demand this maximum error to be zero, but we have to use a tolerance, which depends to the number of elements and the degrees of the polynomials in the finite element basis functions. In Section **??** we reported some experiments with the size of the maximum error. If we want to test that `solver` works for meshes up to $2(20 \times 20)$ elements and cubic Lagrange elements, 10^{-11} is an appropriate tolerance for testing that the maximum error vanishes.

Only three statements are necessary to carry out the unit test. However, we shall embed these statements in software that the testing frameworks pytest and nose can recognize. This means that each unit test must be placed in a function that

- has a name starting with `test_`
- has no arguments
- implements the test as `assert success, msg`

Regarding the last point, `success` is a boolean expression that is `False` if the test fails, and in that case the string `msg` is written to the screen. When the test fails, `assert` raises an `AssertionError` exception in Python, otherwise the statement runs silently. The `msg` string is optional, so `assert success` is the minimal test. In our case, we will do `assert max_error < tol`, where `tol` is the tolerance (10^{-11}) mentioned above.

A proper *test function* for implementing this unit test in the pytest or nose testing frameworks has the following form. Note that we perform the test for different mesh resolutions and degrees of finite elements.

```
def test_solver():
    """Reproduce u=1+x^2+2y^2 to "machine precision"."""
    tol = 1E-11  # This problem's precision
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    for Nx, Ny in [(3,3), (3,5), (5,3), (20,20)]:
        for degree in 1, 2, 3:
            print('solving on 2(%dx%d) mesh with P%d elements'
                  % (Nx, Ny, degree))
            u = solver(f, u0, Nx, Ny, degree)
            # Make a finite element function of the exact u0
            V = u.function_space()
            u0_Function = interpolate(u0, V)  # exact solution
            # Check that dof arrays are equal
            u0_array = u0_Function.vector().array()  # dof values
            max_error = (u0_array - u.vector().array()).max()
            msg = 'max error: %g for 2(%dx%d) mesh and degree=%d' %\
                  (max_error, Nx, Ny, degree)
            assert max_error < tol, msg
```

We can at any time run

Terminal

```
Terminal> py.test -s -v ft04_poisson_func.py
```

and the pytest tool will run all functions `test_*()` in the file and report how the tests go.

We shall make it a habit in this book to encapsulate numerical test problems in unit tests as done above, and we strongly encourage the reader to create similar unit tests whenever a FEniCS solver is implemented. We dare to assert that this is the only serious way do reliable computational science with FEniCS.

Tip: Print messages in test functions

The `assert` statement runs silently when the test passes so users may become uncertain if all the statements in a test function are really executed. A psychological help is to print out something before `assert`

(as we do in the example above) such that it is clear that the test really takes place. (Note that `py.test` needs the `-s` option to show printout from the test functions.)

The next three sections deal with some technicalities about specifying the solution method for linear systems (so that you can solve large problems) and examining array data from the computed solution (so that you can check that the program is correct). These technicalities are scattered around in forthcoming programs. However, the impatient reader who is more interested in seeing the previous program being adapted to a real physical problem, and play around with some interesting visualizations, can safely jump to Section **??**. Information in the intermediate sections can be studied on demand.

1.2 Useful extensions

hpl 3: Need a little intro.

1.2.1 Controlling the solution process

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and recommended method for a few thousand unknowns in the equation system, and may hence be the method of choice in many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and memory demanding in large problems. This fact forces the use of iterative methods, which are faster and require much less memory. The forthcoming text tells you how to advantage of state-of-the-art iterative solution methods in FEniCS.

Setting linear solver parameters. Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite coefficient matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. However, the CG method requires boundary conditions to be implemented in a symmetric way. This is not the case by default, so then a Krylov solver for non-symmetric system, such as GMRES, is a better choice. Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner, so let us try the GMRES-ILU pair:

```
solve(a == L, u, bc)
      solver_parameters={'linear_solver': 'gmres',
                         'preconditioner': 'ilu'})
# Alternative syntax
```

```
solve(a == L, u, bc,
      solver_parameters=dict(linear_solver='gmres',
                             preconditioner='ilu'))
```

Section 1.2.2 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

Linear algebra backend. The actual GMRES and ILU implementations that are brought into action depends on the choice of linear algebra package. FEniCS interfaces several linear algebra packages, called *linear algebra backends* in FEniCS terminology. PETSc is the default choice if FEniCS is compiled with PETSc, otherwise uBLAS. Epetra (Trilinos), Eigen, MTL4 are other supported backends. Which backend to apply can be controlled by setting

```
parameters['linear_algebra_backend'] = backendname
```

where `backendname` is a string, either `'Eigen'`, `'PETSc'`, `'uBLAS'`, `'Epetra'`, or `'MTL4'`. All these backends offer high-quality implementations of both iterative and direct solvers for linear systems of equations.

A common platform for FEniCS users is Ubuntu Linux. The FEniCS distribution for Ubuntu contains PETSc, making this package the default linear algebra backend. The default solver is sparse LU decomposition (`'lu'`), and the actual software that is called is then the sparse LU solver from UMFPACK (which PETSc has an interface to). The available linear algebra backends in a FEniCS installation is listed by

```
list_linear_algebra_backends()
```

The `parameters` database. We will normally like to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be set by accessing the *global parameter database*, which is called `parameters` and which behaves as a nested dictionary. Write

```
info(parameters, verbose=True)
```

to list all parameters and their default values in the database. The nesting of parameter sets is indicated through indentation in the output from `info`. According to this output, the relevant parameter set is named `'krylov_solver'`, and the parameters are set like this:

```
prm = parameters['krylov_solver']  # short form
prm['absolute_tolerance'] = 1E-10
prm['relative_tolerance'] = 1E-6
prm['maximum_iterations'] = 1000
```

Stopping criteria for Krylov solvers usually involve the norm of the residual, which must be smaller than the absolute tolerance parameter *or* smaller than the relative tolerance parameter times the initial residual.

To get a printout of the number of actual iterations to reach the stopping criterion, we can insert

```
set_log_level(PROGRESS)
# or
set_log_level(DEBUG)
```

A message with the equation system size, solver type, and number of iterations arises from specifying the argument `PROGRESS`, while `DEBUG` results in more information, including CPU time spent in the various parts of the matrix assembly and solve process.

We remark that default values for the global parameter database can be defined in an XML file. To generate such a file from the current set of parameters in a program, run

```
File('fenics_parameters.xml') << parameters
```

If a `fenics_parameters.xml` file is found in the directory where a FEniCS program is run, this file is read and used to initialize the `parameters` object. Otherwise, the file `.config/fenics/fenics_parameters.xml` in the user's home directory is read, if it exists. Another alternative is to load the XML (with any name) manually in the program:

```
File('fenics_parameters.xml') >> parameters
```

The XML file can also be in gzip'ed form with the extension `.xml.gz`.

An extended solver function. Let us extend the previous solver function from `ft04_poisson_func.py` such that it also offers the GMRES+ILU preconditioned Krylov solver.

```
from fenics import *

def solver(
    f, u0, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    ):
    """
    Solve -Laplace(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u=u0 (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary
```

```
    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = dot(grad(u), grad(v))*dx
    L = f*v*dx

    # Compute solution
    u = Function(V)

    if linear_solver == 'Krylov':
        prm = parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
                             'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

    solve(a == L, u, bc, solver_parameters=solver_parameters)
    return u

def solver_objects(
    f, u0, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    ):
    """As solver, but use objects for linear variational problem
    and solver."""
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = dot(grad(u), grad(v))*dx
    L = f*v*dx
```

```
    # Compute solution
    u = Function(V)
    problem = LinearVariationalProblem(a, L, u, bc)
    solver  = LinearVariationalSolver(problem)

    if linear_solver == 'Krylov':
        solver.parameters['linear_solver'] = 'gmres'
        solver.parameters['preconditioner'] = 'ilu'
        prm = solver.parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
                             'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

    solver.solve()
    return u
```

This new `solver` function, found in the file `ft05_poisson_iter.py`, replaces the one in `ft04_poisson_func.py`: it has all the functionality of the previous `solver` function, but can also solve the linear system with iterative methods and report the progress of such solvers.

Remark regarding unit tests. Regarding verification of the new `solver` function in terms of unit tests, it turns out that unit testing in a problem where the approximation error vanishes is gets more complicated when we use iterative methods. The problem is to keep the error due to iterative solution smaller than the tolerance used in the verification tests. First of all this means that the tolerances used in the Krylov solvers must be smaller than the tolerance used in the `assert` test, but this is no guarantee to keep the linear solver error this small. For linear elements and small meshes, a tolerance of 10^{-11} works well in the case of Krylov solvers too (using a tolerance 10^{-12} in those solvers. However, as soon as we switch to P2 elements, it is hard to force the linear solver error below 10^{-6}. Consequently, tolerances in tests depend on the numerical methods. The interested reader is referred to the `test_solver` function in `ft05_poisson_iter.py` for details: this test function tests the numerical solution for direct and iterative linear solvers, for different meshes, and different degrees of the polynomials in the finite element basis functions.

1.2.2 Linear solvers and preconditioners

The following solution methods for linear systems can be accessed in FEniCS programs:

Name	Method
`'lu'`	sparse LU factorization (Gaussian elim.)
`'cholesky'`	sparse Cholesky factorization
`'cg'`	Conjugate gradient method
`'gmres'`	Generalized minimal residual method
`'bicgstab'`	Biconjugate gradient stabilized method
`'minres'`	Minimal residual method
`'tfqmr'`	Transpose-free quasi-minimal residual method
`'richardson'`	Richardson method

Possible choices of preconditioners include

Name	Method
`'none'`	No preconditioner
`'ilu'`	Incomplete LU factorization
`'icc'`	Incomplete Cholesky factorization
`'jacobi'`	Jacobi iteration
`'bjacobi'`	Block Jacobi iteration
`'sor'`	Successive over-relaxation
`'amg'`	Algebraic multigrid (BoomerAMG or ML)
`'additive_schwarz'`	Additive Schwarz
`'hypre_amg'`	Hypre algebraic multigrid (BoomerAMG)
`'hypre_euclid'`	Hypre parallel incomplete LU factorization
`'hypre_parasails'`	Hypre parallel sparse approximate inverse
`'ml_amg'`	ML algebraic multigrid

Many of the choices listed above are only offered by a specific backend, so setting the backend appropriately is necessary for being able to choose a desired linear solver or preconditioner. You can also use constructions like

```
prec = 'amg' if has_krylov_solver_preconditioner('amg') \
       else 'default'
```

An up-to-date list of the available solvers and preconditioners in FEniCS can be produced by

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

1.2.3 Linear variational problem and solver objects

The `solve(a == L, u, bc)` call is just a compact syntax alternative to a slightly more comprehensive specification of the variational equation and the solution of the associated linear system. This alternative syntax is used in a lot of FEniCS applications and will also be used later in this tutorial, so we show it already now:

```
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bc)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

Many objects have an attribute `parameters` corresponding to a parameter set in the global `parameters` database, but local to the object. Here, `solver.parameters` play that role. Setting the CG method with ILU preconditioning as solution method and specifying solver-specific parameters can be done like this:

```
solver.parameters['linear_solver'] = 'gmres'
solver.parameters['preconditioner'] = 'ilu'
prm = solver.parameters['krylov_solver'] # short form
prm['absolute_tolerance'] = 1E-7
prm['relative_tolerance'] = 1E-4
prm['maximum_iterations'] = 1000
```

Settings in the global `parameters` database are propagated to parameter sets in individual objects, with the possibility of being overwritten as done above.

The linear variational problem and solver objects as outlined above are incorporated in an alternative solver function, named `solver_objects`, in `ft05_poisson_iter.py`. Otherwise, this function is parallel to the previously shown `solver` function.

1.2.4 Writing out the discrete solution

We have seen how to grab the degrees of freedom array from a finite element function `u`:

```
u_array = u.vector().array()
```

The elements in `u_array` correspond to function values of `u` at nodes in the mesh. Now, a fundamental question is: What are the coordinates of node `i` whose value is `u_array[i]`? To answer this question, we need to understand how to get our hands on the coordinates, and in particular, the numbering of degrees of freedom and the numbering of vertices in the mesh. We start with P1 (1st order Lagrange) elements where all the nodes are vertices in the mesh.

The function `mesh.coordinates()` returns the coordinates of the vertices as a `numpy` array with shape (M,d), M being the number of vertices in the mesh and d being the number of space dimensions:

```
>>> from fenics import *
>>>
>>> mesh = UnitSquareMesh(2, 2)
>>> coor = mesh.coordinates()
>>> coor
array([[ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  0. ],
       [ 0. ,  0.5],
       [ 0.5,  0.5],
       [ 1. ,  0.5],
       [ 0. ,  1. ],
       [ 0.5,  1. ],
       [ 1. ,  1. ]])
```

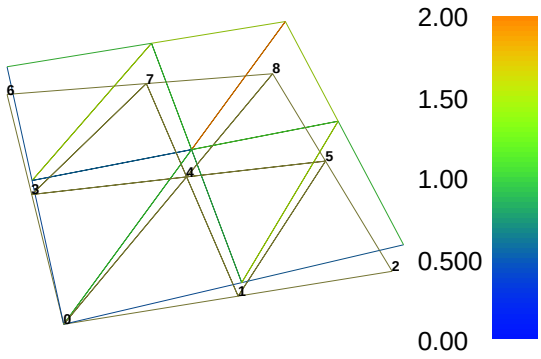
We see from this output that vertices are first numbered along $y = 0$ with increasing x coordinate, then along $y = 0.5$, and so on.

Next we compute a function `u` on this mesh, e.g., the $u = x + y$:

```
>>> V = FunctionSpace(mesh, 'P', 1)
>>> u = interpolate(Expression('x[0]+x[1]'), V)
>>> plot(u, interactive=True)
>>> u_array = u.vector().array()
>>> u_array
array([ 1. ,  0.5,  1.5,  0. ,  1. ,  2. ,  0.5,  1.5,  1. ])
```

We observe that `u_array[0]` is *not* the value of $x + y$ at vertex number 0, since this vertex has coordinates $x = y = 0$. The numbering of the degrees of freedom $U_1, \ldots, U_N$ is obviously not the same as the numbering of the vertices.

In the plot of `u`, type `w` to turn on wireframe instead of fully colored surface, `m` to show the mesh, and then `v` to show the numbering of the vertices.



Already in Section **??** we explained that the vertex values of a `Function` object can be extracted `u.compute_vertex_values()`, which returns an array where element `i` is the value of `u` at vertex `i`:

```
>>> u_at_vertices = u.compute_vertex_values()
>>> for i, x in enumerate(coor):
...     print('vertex %d: u_at_vertices[%d]=%g\tu(%s)=%g' %
...           (i, i, u_at_vertices[i], x, u(x)))
vertex 0: u_at_vertices[0]=0    u([ 0.  0.])=8.46545e-16
vertex 1: u_at_vertices[1]=0.5  u([ 0.5  0. ])=0.5
vertex 2: u_at_vertices[2]=1    u([ 1.  0.])=1
vertex 3: u_at_vertices[3]=0.5  u([ 0.   0.5])=0.5
vertex 4: u_at_vertices[4]=1    u([ 0.5  0.5])=1
vertex 5: u_at_vertices[5]=1.5  u([ 1.   0.5])=1.5
vertex 6: u_at_vertices[6]=1    u([ 0.  1.])=1
vertex 7: u_at_vertices[7]=1.5  u([ 0.5  1. ])=1.5
vertex 8: u_at_vertices[8]=2    u([ 1.  1.])=2
```

Alternatively, we can ask for the mapping from vertex numbering to degrees of freedom numbering in the space V:

```
v2d = vertex_to_dof_map(V)
```

Now, `u_array[v2d[i]]` will give us the value of the degree of freedom in `u` corresponding to vertex `i` (`v2d[i]`). In particular, `u_array[v2d]` is an array with all the elements in the same (vertex numbered) order as `coor`. The inverse map, from degrees of freedom number to vertex number is given by `dof_to_vertex_map(V)`, so `coor[dof_to_vertex_map(V)]` results in an array of all the coordinates in the same order as the degrees of freedom.

For Lagrange elements of degree larger than 1, there are degrees of freedom (nodes) that do not correspond to vertices. **hpl 4**: Anders, is the following true? There is no simple way of getting the coordinates associated with the non-vertex degrees of freedom, so if we want to write out the values of a finite

element solution, the following code snippet does the task at the vertices, and this will work for all kinds of Lagrange elements.

```
def compare_exact_and_numerical_solution(Nx, Ny, degree=1):
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    u = solver(f, u0, Nx, Ny, degree, linear_solver='direct')
    # Grab exact and numerical solution at the vertices and compare
    V = u.function_space()
    u0_Function = interpolate(u0, V)
    u0_at_vertices = u0_Function.compute_vertex_values()
    u_at_vertices = u.compute_vertex_values()
    coor = V.mesh().coordinates()
    for i, x in enumerate(coor):
        print('vertex %2d (%9g,%9g): error=%g'
              % (i, x[0], x[1],
                 u0_at_vertices[i] - u_at_vertices[i]))
        # Could compute u0(x) - u_at_vertices[i] but this
        # is much more expensive and gives more rounding errors
    center = (0.5, 0.5)
    error = u0(center) - u(center)
    print('numerical error at %s: %g' % (center, error))
```

As expected, the error is either identically zero or about 10^{-15} or 10^{-16}.

Cheap vs expensive function evaluation

Given a `Function` object `u`, we can evaluate its values in various ways:

1. `u(x)` for an arbitrary point `x`
2. `u.vector().array()[i]` for degree of freedom number `i`
3. `u.compute_vertex_values()[i]` at vertex number `i`

The first method, though very flexible, is in general very expensive while the other two are very efficient (but limited to certain points).

To demonstrate the use of point evaluations of `Function` objects, we write out the computed `u` at the center point of the domain and compare it with the exact solution:

```
center = (0.5, 0.5)
error = u0(center) - u(center)
print('numerical error at %s: %g' % (center, error)
```

Trying a $2(3\times 3)$ mesh, the output from the previous snippet becomes

```
numerical error at (0.5, 0.5): -0.0833333
```

The discrepancy is due to the fact that the center point is not a node in this particular mesh, but a point in the interior of a cell, and `u` varies linearly over the cell while `u0` is a quadratic function. When the center point is a node, as in a $2(t\times 2)$ or $2(4\times 4)$ mesh, the error is of the order 10^{-15}.

We have seen how to extract the nodal values in a `numpy` array. If desired, we can adjust the nodal values too. Say we want to normalize the solution such that $\max_j U_j = 1$. Then we must divide all U_j values by $\max_j U_j$. The following function performs the task:

```
def normalize_solution(u):
    """Normalize u: return u divided by max(u)."""
    u_array = u.vector().array()
    u_max = u_array.max()
    u_array /= u_max
    u.vector()[:] = u_array
    u.vector().set_local(u_array)  # alternative
    return u
```

That is, we manipulate `u_array` as desired, and then we insert this array into `u`'s `Vector` object. The `/=` operator implies an in-place modification of the object on the left-hand side: all elements of the `u_array` are divided by the value `max_u`. Alternatively, one could write `u_array = u_array/max_u`, which implies creating a new array on the right-hand side and assigning this array to the name `u_array`.

Be careful when manipulating degrees of freedom

A call like `u.vector().array()` returns a *copy* of the data in `u.vector()`. One must therefore never perform assignments like `u.vector.array()[:] = ...`, but instead extract the `numpy` array (i.e., a copy), manipulate it, and insert it back with `u.vector()[:] =` or `u.set_local(...)`.

All the code in this subsection can be found in the file `ft05_poisson_iter.py`.

1.2.5 Parameterizing the number of space dimensions

FEniCS makes it is easy to write a unified simulation code that can operate in 1D, 2D, and 3D. We will conveniently make use of this feature in forthcoming examples. As an appetizer, go back to the introductory programs `ft01_poisson_flat.py` or `ft04_poisson_func.py` and change the mesh construction from `UnitSquareMesh(6, 4)` to `UnitCubeMesh(6, 4, 5)`. Now the domain is the unit cube partitioned into $6 \times 4 \times 5$ boxes, and each box is divided into six tetrahedra-shaped finite elements for computations. Run the program and observe that we can solve a 3D problem without any other modifications (!). The visualization allows you to rotate the cube and observe the function values as colors on the boundary.

Generating a hypercube. The syntax for generating a unit interval, square, or box is different, so we need to encapsulate this part of the code.

Given a list or tuple with the divisions into cells in the various spatial direction, the following function returns the mesh in a d-dimensional problem:

```
def unit_hypercube(divisions, degree):
    mesh_classes = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
    d = len(divisions)
    mesh = mesh_classes[d-1](*divisions)
    V = FunctionSpace(mesh, 'P', degree)
    return V, mesh
```

The construction `mesh_class[d-1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends all the component of the list `divisions` as separate arguments. For example, in a 2D problem where `divisions` has two elements, the statement

```
mesh = mesh_classes[d-1](*divisions)
```

is equivalent to

```
mesh = UnitSquareMesh(divisions[0], divisions[1])
```

Replacing the `Nx` and `Ny` parameters by `divisions` and calling `unit_hypercube` to create the mesh are the two modifications that we need in any of the previously shown `solver` functions to turn them into solvers for d-dimensional problems!

1.2.6 Computing derivatives

In Poisson and many other problems, the gradient of the solution is of interest. The computation is in principle simple: since $u = \sum_{j=1}^{N} U_j \phi_j$, we have that

$$\nabla u = \sum_{j=1}^{N} U_j \nabla \phi_j \,.$$

Given the solution variable `u` in the program, its gradient is obtained by `grad(u)` or `grad(u)`. However, the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the ϕ_j has discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, u is linear over each cell, and the numerical ∇u becomes a piecewise constant vector field. On the contrary, the exact gradient is continuous. For visualization and data analysis purposes we often want the computed gradient to be a continuous vector field. Typically, we want each component of ∇u to be represented in the same way as u itself. To this end, we can project the components of ∇u onto the same function space as we used for u. This means that we solve $w = \nabla u$ approximately by a finite

element method, using the same elements for the components of w as we used for u. This process is known as *projection*.

Not surprisingly, projection is a so common operation in finite element programs that FEniCS has a function for doing the task: `project(q, W)`, which returns the projection of some `Function` or `Expression` object named `q` onto the `FunctionSpace` (if `q` is scalar) or `VectorFunctionSpace` (if `q` is vector-valued) named `W`. Specifically, in our case where `u` is computed and we want to project the vector-valued `grad(u)` onto the `VectorFunctionSpace` where each component has the same `Function` space as `u`:

```
V = u.function_space()
degree = u.ufl_element().degree()
W = VectorFunctionSpace(V.mesh(), 'P', degree)

grad_u = project(grad(u), W)
```

Figure 1.1 shows example of how such a smoothed `gradu(u)` vector field is visualized.

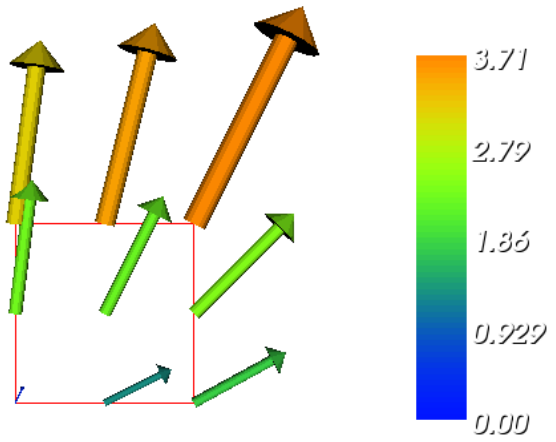


Fig. 1.1 Example of visualizing the vector field ∇u by arrows at the nodes.

The applications of projection are many, including turning discontinuous gradient fields into continuous ones, comparing higher- and lower-order function approximations, and transforming a higher-order finite element solution down to a piecewise linear field, which is required by many visualization packages.

The scalar component fields of the gradient can be extracted as separate fields and, e.g., visualized:

```
grad_u_x, grad_u_y = grad_u.split(deepcopy=True)
plot(grad_u_x, title='x-component of grad(u)')
plot(grad_u_y, title='y-component of grad(u)')
```

The `deepcopy=True` argument signifies a *deep copy*, which is a general term in computer science implying that a copy of the data is returned. (The opposite, `deepcopy=False`, means a *shallow copy*, where the returned objects are just pointers to the original data.)

The `grad_u_x` and `grad_u_y` variables behave as `Function` objects. In particular, we can extract the underlying arrays of nodal values by

```
grad_u_x_array = grad_u_x.vector().array()
grad_u_y_array = grad_u_y.vector().array()
```

The degrees of freedom of the `grad_u` vector field can also be reached by

```
grad_u_array = grad_u.vector().array()
```

but this is a flat `numpy` array where the degrees of freedom for the x component of the gradient is stored in the first part, then the degrees of freedom of the y component, and so on. This is less convenient to work with.

The function `gradient(u)` in `ft05_poisson_iter.py` returns a projected (smoothed) ∇u vector field, given some finite element function `u`:

```
def gradient(u):
    """Return grad(u) projected onto same space as u."""
    V = u.function_space()
    mesh = V.mesh()
    V_g = VectorFunctionSpace(mesh, 'P', 1)
    grad_u = project(grad(u), V_g)
    grad_u.rename('grad(u)', 'continuous gradient field')
    return grad_u
```

Examining the arrays with vertex values of `grad_u_x` and `grad_u_y` quickly reveals that the computed `grad_u` field does not equal the exact gradient $(2x, 4y)$ in this particular test problem where $u = 1 + x^2 + 2y^2$. There are inaccuracies at the boundaries, arising from the approximation problem for w. Increasing the mesh resolution shows, however, that the components of the gradient vary linearly as $2x$ and $4y$ in the interior of the mesh (i.e., as soon as we are one element away from the boundary). The `application_test_gradient` function in `ft05_poisson_iter.py` performs some experiments.

Detour: Manual projection.

Although you will always use `project` to project a finite element function, it can be constructive this point in the tutorial to formulate the projection mathematically and implement its steps manually in FEniCS.

Looking at the component $\partial u/\partial x$ of the gradient, we project the (discrete) derivative $\sum_j U_j \partial\phi_j/\partial x$ onto a function space with basis $\phi_1, \phi_2, \ldots$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \phi_j$, for suitable (new) coefficients $\bar{U}_j$.

The variational problem for w reads: find $w \in V^{(\mathrm{g})}$ such that

$$a(w, v) = L(v) \quad \forall v \in \hat{V^{(\mathrm{g})}}, \tag{1.1}$$

where

$$a(w, v) = \int_\Omega w \cdot v \,\mathrm{d}x, \tag{1.2}$$

$$L(v) = \int_\Omega \nabla u \cdot v \,\mathrm{d}x\,. \tag{1.3}$$

The function spaces $V^{(\mathrm{g})}$ and $\hat{V^{(\mathrm{g})}}$ (with the superscript g denoting "gradient") are vector versions of the function space for u, with boundary conditions removed (if V is the space we used for u, with no restrictions on boundary values, $V^{(\mathrm{g})} = \hat{V^{(\mathrm{g})}} = [V]^d$, where d is the number of space dimensions). For example, if we used piecewise linear functions on the mesh to approximate u, the variational problem for w corresponds to approximating each component field of w by piecewise linear functions.

The variational problem for the vector field w, called `grad_u` in the code, is easy to solve in FEniCS:

```
V_g = VectorFunctionSpace(mesh, 'P', 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

a = dot(w, v)*dx
L = dot(grad(u), v)*dx
grad_u = Function(V_g)
solve(a == L, grad_u)

plot(grad_u, title='grad(u)')
```

The boundary condition argument to `solve` is dropped since there are no essential boundary conditions in this problem. The new thing is basically that we work with a `VectorFunctionSpace`, since the unknown is now a vector field, instead of the `FunctionSpace` object for scalar fields.

1.2.7 A variable-coefficient Poisson problem

Suppose we have a variable coefficient $p(x,y)$ in the Laplace operator, as in the boundary-value problem

$$\begin{aligned} -\nabla\cdot[p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \quad \text{on } \partial\Omega\,. \end{aligned} \tag{1.4}$$

We shall quickly demonstrate that this simple extension of our model problem only requires an equally simple extension of the FEniCS program.

Test problem. Let us continue to use our favorite solution $u(x,y) = 1+x^2+2y^2$ and then prescribe $p(x,y) = x+y$. It follows that $u_0(x,y) = 1+x^2+2y^2$ and $f(x,y) = -8x-10y$.

Modifications of the PDE solver. What are the modifications we need to do in the previously shown codes to incorporate the variable coefficient p? from Section 1.2.4?

- `solver` must take `p` as argument,
- `f` in our test problem must be an `Expression` since it is no longer a constant,
- a new `Expression p` must be defined for the variable coefficient,
- the formula for $a(u,v)$ in the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE by a test function v and integrating by parts now results in

$$\int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s = \int_\Omega fv\,\mathrm{d}x\,.$$

The function spaces for u and v are the same as in Section **??**, implying that the boundary integral vanishes since $v=0$ on $\partial\Omega$ where we have Dirichlet conditions. The weak form $a(u,v) = L(v)$ then has

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{1.5}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x\,. \tag{1.6}$$

In the code for solving $-\nabla^2 u = f$ we must replace

```
a = dot(grad(u), grad(v))*dx
```

by

```
a = p*dot(grad(u), grad(v))*dx
```

to solve $-\nabla\cdot(p\nabla u) = f$. Moreover, the definitions of `p` and `f` in the test problem read

```
p = Expression('x[0] + x[1]')
f = Expression('-8*x[0] - 10*x[1]')
```

No additional modifications are necessary. The file `ft06_poisson_vc.py` (variable-coefficient Poisson problem in 2D) is a copy of `ft05_poisson_iter.py` with the mentioned changes incorporated. Observe that $p = 1$ recovers the original problem in `ft05_poisson_iter.py`.

You can run it and confirm that it recovers the exact u at the nodes.

Modifications of the flux computations. The flux $-p\nabla u$ may be of particular interest in variable-coefficient Poisson problems as it often has an interesting physical significance. As explained in Section 1.2.6, we normally want the piecewise discontinuous flux or gradient to be approximated by a continuous vector field, using the same elements as used for the numerical solution u. The approximation now consists of solving $w = -p\nabla u$ by a finite element method: find $w \in V^{(\mathrm{g})}$ such that

$$a(w,v) = L(v) \quad \forall v \in \hat{V^{(\mathrm{g})}}, \tag{1.7}$$

where

$$a(w,v) = \int_\Omega w\cdot v\,\mathrm{d}x, \tag{1.8}$$

$$L(v) = \int_\Omega (-p\nabla u)\cdot v\,\mathrm{d}x\,. \tag{1.9}$$

This problem is identical to the one in Section 1.2.6, except that p enters the integral in L.

The relevant Python statement for computing the flux field take the form

```
flux = project(-p*grad(u),
               VectorFunctionSpace(mesh, 'P', degreee))
```

An appropriate function for computing the flux based on `u` and `p` is

```
def flux(u, p):
    """Return p*grad(u) projected onto same space as u."""
    V = u.function_space()
    mesh = V.mesh()
    degree = u.ufl_element().degree()
    V_g = VectorFunctionSpace(mesh, 'P', degree)
    flux_u = project(-p*grad(u), V_g)
    flux_u.rename('flux(u)', 'continuous flux field')
    return flux_u

def application_test_flux(Nx=6, Ny=4):
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
```

```
    p = Expression('x[0] + x[1]')
    f = Expression('-8*x[0] - 10*x[1]')
    u = solver(p, f, u0, Nx, Ny, 1, linear_solver='direct')
    u.rename('u', 'solution')
    flux_u = flux(u, p)
    # Grab each component as a scalar field
    flux_u_x, flux_u_y = flux_u.split(deepcopy=True)
    flux_u_x.rename('flux(u)_x', 'x-component of flux(u)')
    flux_u_y.rename('flux(u)_y', 'y-component of flux(u)')
    plot(u, title=u.label())
    plot(flux_u,   title=flux_u.label())
    plot(flux_u_x, title=flux_u_x.label())
    plot(flux_u_y, title=flux_u_y.label())

    u_exact = lambda x, y: 1 + x**2 + 2*y**2
    flux_x_exact = lambda x, y: -(x+y)*2*x
    flux_y_exact = lambda x, y: -(x+y)*4*y

    coor = u.function_space().mesh().coordinates()
    if len(coor) < 50:
        # Quite large errors for coarse meshes, but the error
        # decreases with increasing resolution
        for i, value in enumerate(flux_u_x.compute_vertex_values()):
            print('vertex %d, %s, -p*u_x=%g, error=%g' %
                  (i, tuple(coor[i]), value,
                   flux_x_exact(*coor[i]) - value))
        for i, value in enumerate(flux_u_y.compute_vertex_values()):
            print('vertex %d, %s, -p*u_y=%g, error=%g' %
                  (i, tuple(coor[i]), value,
                   flux_y_exact(*coor[i]) - value))
    else:
        # Compute integrated L2 error of the flux components
        # (Will this work for unstructured mesh? Need to think about that)
        xv = coor.T[0]
        yv = coor.T[1]

def compute_errors(u, u_exact):
    """Compute various measures of the error u - u_exact, where
    u is a finite element Function and u_exact is an Expression."""

    # Compute error norm (for very small errors, the value can be
    # negative so we run abs(assemble(error)) to avoid failure in sqrt

    V = u.function_space()

    # Function - Expression
    error = (u - u_exact)**2*dx
    E1 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_e onto the same space as u:
    u_e = interpolate(u_exact, V)
    error = (u - u_e)**2*dx
    E2 = sqrt(abs(assemble(error)))
```

```
    # Explicit interpolation of u_exact to higher-order elements,
    # u will also be interpolated to the space Ve before integration
    Ve = FunctionSpace(V.mesh(), 'P', 5)
    u_e = interpolate(u_exact, Ve)
    error = (u - u_e)**2*dx
    E3 = sqrt(abs(assemble(error)))

    # fenics.errornorm interpolates u and u_e to a space with
    # given degree, and creates the error field by subtracting
    # the degrees of freedom, then the error field is integrated
    # TEMPORARY BUG - doesn't accept Expression for u_e
    #E4 = errornorm(u_e, u, normtype='l2', degree=3)
    # Manual implementation errornorm to get around the bug:
    def errornorm(u_exact, u, Ve):
        u_Ve = interpolate(u, Ve)
        u_e_Ve = interpolate(u_exact, Ve)
        e_Ve = Function(Ve)
        # Subtract degrees of freedom for the error field
        e_Ve.vector()[:] = u_e_Ve.vector().array() - u_Ve.vector().array()
        # More efficient computation (avoids the rhs array result above)
        #e_Ve.assign(u_e_Ve)                     # e_Ve = u_e_Ve
        #e_Ve.vector().axpy(-1.0, u_Ve.vector())  # e_Ve += -1.0*u_Ve
        error = e_Ve**2*dx(Ve.mesh())
        return sqrt(abs(assemble(error))), e_Ve
    E4, e_Ve = errornorm(u_exact, u, Ve)

    # Infinity norm based on nodal values
    u_e = interpolate(u_exact, V)
    E5 = abs(u_e.vector().array() - u.vector().array()).max()

    # H1 seminorm
    error = dot(grad(e_Ve), grad(e_Ve))*dx
    E6 = sqrt(abs(assemble(error)))

    # Collect error measures in a dictionary with self-explanatory keys
    errors = {'u - u_exact': E1,
              'u - interpolate(u_exact,V)': E2,
              'interpolate(u,Ve) - interpolate(u_exact,Ve)': E3,
              'errornorm': E4,
              'infinity norm (of dofs)': E5,
              'grad(error) H1 seminorm': E6}

    return errors

def convergence_rate(u_exact, f, u0, p, degrees,
                     n=[2**(k+3) for k in range(5)]):
    """
    Compute convergence rates for various error norms for a
    sequence of meshes with Nx=Ny=b and P1, P2, ...,
    Pdegrees elements. Return rates for two consecutive meshes:
    rates[degree][error_type] = r0, r1, r2, ...
    """

    h = {}  # Discretization parameter, h[degree][experiment]
```

```
    E = {}  # Error measure(s), E[degree][experiment][error_type]
    P_degrees = 1,2,3,4
    num_meshes = 5

    # Perform experiments with meshes and element types
    for degree in P_degrees:
        n = 4   # Coarsest mesh division
        h[degree] = []
        E[degree] = []
        for i in range(num_meshes):
            n *= 2
            h[degree].append(1.0/n)
            u = solver(p, f, u0, n, n, degree,
                       linear_solver='direct')
            errors = compute_errors(u, u_exact)
            E[degree].append(errors)
            print('2*(%dx%d) P%d mesh, %d unknowns, E1=%g' %
                  (n, n, degree, u.function_space().dim(),
                   errors['u - u_exact']))
    # Convergence rates
    from math import log as ln  # log is a fenics name too
    error_types = list(E[1][0].keys())
    rates = {}
    for degree in P_degrees:
        rates[degree] = {}
        for error_type in sorted(error_types):
            rates[degree][error_type] = []
            for i in range(num_meshes):
                Ei   = E[degree][i][error_type]
                Eim1 = E[degree][i-1][error_type]
                r = ln(Ei/Eim1)/ln(h[degree][i]/h[degree][i-1])
                rates[degree][error_type].append(round(r,2))
    return rates

def convergence_rate_sin():
    """Compute convergence rates for u=sin(x)*sin(y) solution."""
    omega = 1.0
    u_exact = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                         omega=omega)
    f = 2*omega**2*pi**2*u_exact
    u0 = Constant(0)
    p = Constant(1)
    # Note: P4 for n>=128 seems to break down
    rates = convergence_rates(u_exact, f, u0, p, degrees=4,
                              n=[2**(k+3) for k in range(5)])
    # Print rates
    print('\n\n')
    for error_type in error_types:
        print(error_type)
        for degree in P_degrees:
            print('P%d: %s' %
                  (degree, str(rates[degree][error_type])[1:-1]))

def structured_mesh(u, divisions):
```

```
    """Represent u on a structured mesh."""
    # u must have P1 elements, otherwise interpolate to P1 elements
    u2 = u if u.ufl_element().degree() == 1 else \
         interpolate(u, FunctionSpace(mesh, 'P', 1))
    mesh = u.function_space().mesh()
    from BoxField import fenics_function2BoxField
    u_box = fenics_function2BoxField(
        u2, mesh, divisions, uniform_mesh=True)
    return u_box

def application_structured_mesh(model_problem=1):
    if model_problem == 1:
        # Numerical solution is exact
        u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
        p = Expression('x[0] + x[1]')
        f = Expression('-8*x[0] - 10*x[1]')
        flux_u_x_exact = lambda x, y: -(x + y)*2*x
        nx = 6;  ny = 4
    elif model_problem == 2:
        # Mexican hat solution
        from sympy import exp, sin, pi  # for use in math formulas
        import sympy as sym
        H = lambda x: exp(-16*(x-0.5)**2)*sin(3*pi*x)
        x, y = sym.symbols('x[0], x[1]')
        u = H(x)*H(y)
        u_c = sym.printing.ccode(u)
        # '-exp(-16*pow(x - 0.5, 2) - 16*pow(y - 0.5, 2))*'
        # 'sin(3*M_PI*x)*sin(3*M_PI*y)'
        u_c = u_c.replace('M_PI', 'DOLFIN_PI')
        print('u in C:', u_c)
        u0 = Expression(u_c)

        p = 1  # Don't use Constant(1) here (!)
        f = sym.diff(-p*sym.diff(u, x), x) + \
            sym.diff(-p*sym.diff(u, y), y)
        f = sym.simplify(f)
        f_c = sym.printing.ccode(f)
        f_c = f_c.replace('M_PI', 'DOLFIN_PI')
        f = Expression(f_c)
        flux_u_x_exact = sym.lambdify([x, y], -p*sym.diff(u, x),
                                      modules='numpy')
        print('f in C:', f_c)
        p = Constant(1)
        nx = 22;  ny = 22

    u = solver(p, f, u0, nx, ny, 1, linear_solver='direct')
    u_box = structured_mesh(u, (nx, ny))
    u_ = u_box.values  # numpy array
    X = 0;  Y = 1      # for indexing in x and y direction

    # Iterate over 2D mesh points (i,j)
    print('u_ is defined on a structured mesh with %s points'
          % str(u_.shape))
    if u.function_space().dim() < 100:
```

```
        for j in range(u_.shape[1]):
            for i in range(u_.shape[0]):
                print('u[%d,%d]=u(%g,%g)=%g' %
                      (i, j,
                       u_box.grid.coor[X][i], u_box.grid.coor[X][j],
                       u_[i,j]))

    # Make surface plot
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib import cm
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    cv = u_box.grid.coorv  # vectorized mesh coordinates
    ax.plot_surface(cv[X], cv[Y], u_, cmap=cm.coolwarm,
                    rstride=1, cstride=1)
    plt.title('Surface plot of solution')
    plt.savefig('tmp0.png'); plt.savefig('tmp0.pdf')

    # Make contour plot
    fig = plt.figure()
    ax = fig.gca()
    cs = ax.contour(cv[X], cv[Y], u_, 7)  # 7 levels
    plt.clabel(cs)  # add labels to contour lines
    plt.axis('equal')
    plt.title('Contour plot of solution')
    plt.savefig('tmp1.png'); plt.savefig('tmp1.pdf')

    # Plot u along a line y=const and compare with exact solution
    start = (0, 0.4)
    x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
    u_e_val = [u0((x_, y_fixed)) for x_ in x]

    plt.figure()
    plt.plot(x, u_val, 'r-')
    plt.plot(x, u_e_val, 'bo')
    plt.legend(['P1 elements', 'exact'], loc='best')
    plt.title('Solution along line y=%g' % y_fixed)
    plt.xlabel('x');  plt.ylabel('u')
    plt.savefig('tmp2.png'); plt.savefig('tmp2.pdf')

    flux_u = flux(u, p)
    flux_u_x, flux_u_y = flux_u.split(deepcopy=True)

    # Plot the numerical and exact flux along the same line
    flux2_x = flux_u_x if flux_u_x.ufl_element().degree() == 1 \
              else interpolate(flux_x,
                   FunctionSpace(u.function_space().mesh(),
                                 'P', 1))
    flux_u_x_box = structured_mesh(flux_u_x, (nx,ny))
    x, flux_u_val, y_fixed, snapped = \
       flux_u_x_box.gridline(start, direction=X)
    y = y_fixed
```

```
    plt.figure()
    plt.plot(x, flux_u_val, 'r-')
    plt.plot(x, flux_u_x_exact(x, y_fixed), 'bo')
    plt.legend(['P1 elements', 'exact'], loc='best')
    plt.title('Flux along line y=%g' % y_fixed)
    plt.xlabel('x');  plt.ylabel('u')
    plt.savefig('tmp3.png'); plt.savefig('tmp3.pdf')

    plt.show()

def solver_linalg(
    p, f, u0, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    assembly='variational', # or 'matvec' or 'system'
    start_vector='zero',    # or 'random'
    ):
    """
    Solve -div(p*grad(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u=u0 (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = dot(p*grad(u), grad(v))*dx
    L = f*v*dx

    # Compute solution
    u = Function(V)
    U = u.vector()
    if initial_guess == 'random':
        import numpy as np
        np.random.seed(10)  # for testing
        U[:] = numpy.random.uniform(-100, 100, n)

    if assembly == 'variational':
        if linear_solver == 'Krylov':
            prm = parameters['krylov_solver'] # short form
            prm['absolute_tolerance'] = abs_tol
            prm['relative_tolerance'] = rel_tol
```

```
            prm['maximum_iterations'] = max_iter
            prm['nonzero_initial_guess'] = True
            print(parameters['linear_algebra_backend'])
            set_log_level(log_level)
            if dump_parameters:
                info(parameters, True)
            solver_parameters = {'linear_solver': 'gmres',
                                 'preconditioner': 'ilu'}
        else:
            solver_parameters = {'linear_solver': 'lu'}

        solve(a == L, u, bc, solver_parameters=solver_parameters)
        A = None # Cannot return cofficient matrix
    else:
        if assembly == 'matvec':
            A = assemble(a)
            b = assemble(L)
            bc.apply(A, b)
            if linear_solver == 'direct':
                solve(A, U, b)
            else:
                solver = KrylovSolver('gmres', 'ilu')
                prm = solver.parameters
                prm['absolute_tolerance'] = abs_tol
                prm['relative_tolerance'] = rel_tol
                prm['maximum_iterations'] = max_iter
                prm['nonzero_initial_guess'] = True
                solver.solve(A, U, b)
        elif assembly == 'system':
            A, b = assemble_system(a, L, [bc])
            if linear_solver == 'direct':
                solve(A, U, b)
            else:
                solver = KrylovSolver('cg', 'ilu')
                prm = solver.parameters
                prm['absolute_tolerance'] = abs_tol
                prm['relative_tolerance'] = rel_tol
                prm['maximum_iterations'] = max_iter
                prm['nonzero_initial_guess'] = True
                solver.solve(A, U, b)
    return u, A

def application_linalg():
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    p = Expression('x[0] + x[1]')
    f = Expression('-8*x[0] - 10*x[1]')
    meshes = [2, 8, 32, 128]
    for n in meshes:
        for assembly in 'variational', 'matvec', 'system':
            print('--- %dx%d mesh, %s assembly ---' % (n, n, assembly))
            u, A = solver_linalg(
                p, f, u0, n, n, linear_solver='Krylov',
                assembly=assembly)
            if A is not None and u.function_space().dim() < 10:
```

```
                import numpy as np
                np.set_printoptions(precision=2)
                print('A: %s assembly\n' % assembly, A.array())

def solver_bc(
    p, f,                   # Coefficients in the PDE
    boundary_conditions,    # Dict of boundary conditions
    Nx, Ny,                 # Cell division of the domain
    degree=1,               # Polynomial degree
    subdomains=[],          # List of SubDomain objects in domain
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    debug=False,
    ):
    """
    Solve -div(p*grad(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and Dirichlet, Neumann, or Robin
    conditions on the boundary. Piecewise constant p over subdomains
    are also allowed.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    tol = 1E-14

    # Subdomains in the domain?
    import numpy as np
    if subdomains:
        # subdomains is list of SubDomain objects,
        # p is array of corresponding constant values of p
        # in each subdomain
        if not isinstance(p, (list, tuple, np.ndarray)):
            raise TypeError(
                'p must be array if we have sudomains, not %s'
                % type(p))
        materials = CellFunction('size_t', mesh)
        materials.set_all(0)  # "the rest"
        for m, subdomain in enumerate(subdomains[1:], 1):
            subdomain.mark(materials, m)

        p_values = p
        V0 = FunctionSpace(mesh, 'DG', 0)
        p  = Function(V0)
        help = np.asarray(materials.array(), dtype=np.int32)
        p.vector()[:] = np.choose(help, p_values)
    else:
        if not isinstance(p, (Expression, Constant)):
            raise TypeError(
                'p is type %s, must be Expression or Constant'
```

```
                % type(p))

    # Boundary subdomains
    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
    boundary_parts = FacetFunction('size_t', 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)
    # boundary_parts.array() is a numpy array

    ds = Measure('ds', domain=mesh, subdomain_data=boundary_parts)

    # boundary_conditions is a dict of dicts:
    # {0: {'Dirichlet': u0},
    #  1: {'Robin': (r, s)},
    #  2: {'Neumann: g}},
    #  3: {'Neumann', 0}}

    bcs = []  # List of Dirichlet conditions
    for n in boundary_conditions:
        if 'Dirichlet' in boundary_conditions[n]:
            bcs.append(
                DirichletBC(V, boundary_conditions[n]['Dirichlet'],
                            boundary_parts, n))

    if debug:
        # Print the vertices that are on the boundaries
        coor = mesh.coordinates()
        for x in coor:
            if bx0.inside(x, True): print('%s is on x=0' % x)
            if bx1.inside(x, True): print('%s is on x=1' % x)
            if by0.inside(x, True): print('%s is on y=0' % x)
```

```
            if by1.inside(x, True): print('%s is on y=1' % x)


        # Print the Dirichlet conditions
        print('No of Dirichlet conditions:', len(bcs))
        d2v = dof_to_vertex_map(V)
        for bc in 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()))))

    # Collect Neumann integrals
    u = TrialFunction(V)
    v = TestFunction(V)

    Neumann_integrals = []
    for n in boundary_conditions:
        if 'Neumann' in boundary_conditions[n]:
            if boundary_conditions[n]['Neumann'] != 0:
                g = boundary_conditions[n]['Neumann']
                Neumann_integrals.append(g*v*ds(n))

    # Collect Robin integrals
    Robin_a_integrals = []
    Robin_L_integrals = []
    for n in boundary_conditions:
        if 'Robin' in boundary_conditions[n]:
            r, s = boundary_conditions[n]['Robin']
            Robin_a_integrals.append(r*u*v*ds(n))
            Robin_L_integrals.append(r*s*v*ds(n))

    # Simpler Robin integrals
    Robin_integrals = []
    for n in boundary_conditions:
        if 'Robin' in boundary_conditions[n]:
            r, s = boundary_conditions[n]['Robin']
            Robin_integrals.append(r*(u-s)*v*ds(n))

    # Define variational problem, solver_bc
    a = dot(p*grad(u), grad(v))*dx + \
        sum(Robin_a_integrals)
    L = f*v*dx - sum(Neumann_integrals) + sum(Robin_L_integrals)

    # Simpler variational formulation
    F = dot(p*grad(u), grad(v))*dx + \
        sum(Robin_integrals) - f*v*dx + sum(Neumann_integrals)
    a, L = lhs(F), rhs(F)

    # Compute solution
    u = Function(V)
```

```
    if linear_solver == 'Krylov':
        prm = parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
                             'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

    solve(a == L, u, bcs, solver_parameters=solver_parameters)
    return u, p  # Note: p may be modified (Function on V0)

def application_bc_test():
    # Define manufactured solution in sympy and derive f, g, etc.
    import sympy as sym
    x, y = sym.symbols('x[0] x[1]')  # UFL needs x[0] for x etc.
    u = 1 + x**2 + 2*y**2
    f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)  # -Laplace(u)
    f = sym.simplify(f)
    u_00 = u.subs(x, 0)  # x=0 boundary
    u_01 = u.subs(x, 1)  # x=1 boundary
    g = -sym.diff(u, y).subs(y, 1)  # x=1 boundary, du/dn=-du/dy
    r = 1000 # any function can go here
    s = u

    # Turn to C/C++ code for UFL expressions
    f = sym.printing.ccode(f)
    u_00 = sym.printing.ccode(u_00)
    u_01 = sym.printing.ccode(u_01)
    g = sym.printing.ccode(g)
    r = sym.printing.ccode(r)
    s = sym.printing.ccode(s)
    print('Test problem (C/C++):\nu = %s\nf = %s' % (u, f))
    print('u_00: %s\nu_01: %s\ng = %s\nr = %s\ns = %s' %
          (u_00, u_01, g, r, s))

    # Turn into FEniCS objects
    u_00 = Expression(u_00)
    u_01 = Expression(u_01)
    f = Expression(f)
    g = Expression(g)
    r = Expression(r)
    s = Expression(s)
    u_exact = Expression(sym.printing.ccode(u))

    boundary_conditions = {
        0: {'Dirichlet': u_00},   # x=0
        1: {'Dirichlet': u_01},   # x=1
        2: {'Robin': (r, s)},     # y=0
```

```
        3: {'Neumann': g}}        # y=1

    p = Constant(1)
    Nx = Ny = 2
    u, p = solver_bc(
        p, f, boundary_conditions, Nx, Ny, degree=1,
        linear_solver='direct',
        debug=2*Nx*Ny < 50,  # for small problems only
        )

    # Compute max error in infinity norm
    u_e = interpolate(u_exact, u.function_space())
    import numpy as np
    max_error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
    print('Max error:', max_error)

    # Print numerical and exact solution at the vertices
    if u.function_space().dim() < 50:  # (small problems only)
        u_e_at_vertices = u_e.compute_vertex_values()
        u_at_vertices = u.compute_vertex_values()
        coor = u.function_space().mesh().coordinates()
        for i, x in enumerate(coor):
            print('vertex %2d (%9g,%9g): error=%g %g vs %g'
                  % (i, x[0], x[1],
                     u_e_at_vertices[i] - u_at_vertices[i],
                     u_e_at_vertices[i], u_at_vertices[i]))

def test_solvers_bc():
    """Reproduce u=1+x^2+2y^2 to with different solvers."""
    tol = 3E-12  # Appropriate tolerance for these tests (P2, 20x20 mesh)
    import sympy as sym
    x, y = sym.symbols('x[0] x[1]')
    u = 1 + x**2 + 2*y**2
    f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)
    f = sym.simplify(f)
    u_00 = u.subs(x, 0)  # x=0 boundary
    u_01 = u.subs(x, 1)  # x=1 boundary
    g = -sym.diff(u, y).subs(y, 1)  # x=1 boundary
    r = 1000 # arbitrary function can go here
    s = u

    # Turn to C/C++ code for UFL expressions
    f = sym.printing.ccode(f)
    u_00 = sym.printing.ccode(u_00)
    u_01 = sym.printing.ccode(u_01)
    g = sym.printing.ccode(g)
    r = sym.printing.ccode(r)
    s = sym.printing.ccode(s)
    print('Test problem (C/C++):\nu = %s\nf = %s' % (u, f))
    print('u_00: %s\nu_01: %s\ng = %s\nr = %s\ns = %s' %
          (u_00, u_01, g, r, s))

    # Turn into FEniCS objects
```

```
    u_00 = Expression(u_00)
    u_01 = Expression(u_01)
    f = Expression(f)
    g = Expression(g)
    r = Expression(r)
    s = Expression(s)
    u_exact = Expression(sym.printing.ccode(u))

    boundary_conditions = {
        0: {'Dirichlet': u_00},
        1: {'Dirichlet': u_01},
        2: {'Robin': (r, s)},
        3: {'Neumann': g}}

    p = Constant(1)

    for Nx, Ny in [(3,3), (3,5), (5,3), (20,20)]:
        for degree in 1, 2, 3:
            for linear_solver in ['direct']:
                print('solving on 2(%dx%dx) mesh with P%d elements'
                      % (Nx, Ny, degree)),
                print(' %s solver, %s function' %
                      (linear_solver, solver_func.__name__))
                u, p = solver_bc(
                    p, f, boundary_conditions, Nx, Ny, degree,
                linear_solver=linear_solver,
                    abs_tol=0.1*tol,
                    rel_tol=0.1*tol)
                # Make a finite element function of the exact u0
                V = u.function_space()
                u_e_Function = interpolate(u_exact, V)  # exact solution
                # Check that dof arrays are equal
                u_e_array = u_e_Function.vector().array()  # dof values
                max_error = (u_e_array - u.vector().array()).max()
                msg = 'max error: %g for 2(%dx%d) mesh, degree=%d,'\
                      ' %s solver, %s' % \
                      (max_error, Nx, Ny, degree, linear_solver,
                       solver_func.__name__)
                print(msg)
                assert max_error < tol, msg

def application_bc_test_2mat():
    tol = 1E-14  # Tolerance for coordinate comparisons

    class Omega0(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] <= 0.5+tol

    class Omega1(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] >= 0.5-tol

    subdomains = [Omega0(), Omega1()]
    p_values = [2.0, 13.0]
```

```
    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])


    boundary_conditions = {
        0: {'Neumann': 0},
        1: {'Neumann': 0},
        2: {'Dirichlet': Constant(0)}, # y=0
        3: {'Dirichlet': Constant(1)}, # y=1
        }

    f = Constant(0)
    Nx = Ny = 2
    u, p = solver_bc(
        p_values, f, boundary_conditions, Nx, Ny, degree=1,
        linear_solver='direct', subdomains=subdomains,
        debug=2*Nx*Ny < 50,  # for small problems only
        )

    # Compute max error in infinity norm
    u_e = interpolate(u_exact, u.function_space())
    import numpy as np
    max_error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
    print('Max error:', max_error)

    # Print numerical and exact solution at the vertices
    if u.function_space().dim() < 50:  # (small problems only)
        u_e_at_vertices = u_e.compute_vertex_values()
        u_at_vertices = u.compute_vertex_values()
        coor = u.function_space().mesh().coordinates()
        for i, x in enumerate(coor):
            print('vertex %2d (%9g,%9g): error=%g %g vs %g'
                  % (i, x[0], x[1],
                     u_e_at_vertices[i] - u_at_vertices[i],
                     u_e_at_vertices[i], u_at_vertices[i]))

def test_solvers_bc_2mat():
    tol = 2E-13  # Tolerance for comparisons

    class Omega0(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] <= 0.5+tol

    class Omega1(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] >= 0.5-tol

    subdomains = [Omega0(), Omega1()]
    p_values = [2.0, 13.0]
    boundary_conditions = {
```

```
        0: {'Neumann': 0},
        1: {'Neumann': 0},
        2: {'Dirichlet': Constant(0)}, # y=0
        3: {'Dirichlet': Constant(1)}, # y=1
        }

    f = Constant(0)
    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])

    for Nx, Ny in [(2,2), (2,4), (8,4)]:
        for degree in 1, 2, 3:
            u, p = solver_bc(
                p_values, f, boundary_conditions, Nx, Ny, degree,
                linear_solver='direct', subdomains=subdomains,
                debug=False)

            # Compute max error in infinity norm
            u_e = interpolate(u_exact, u.function_space())
            import numpy as np
            max_error = np.abs(u_e.vector().array() -
                               u.vector().array()).max()
            assert max_error < tol, 'max error: %g' % max_error

def application_flow_around_circle(obstacle='rectangle'):
    tol = 1E-14  # Tolerance for coordinate comparisons

    class Circle(SubDomain):
        def inside(self, x, on_boundary):
            return ((x[0]-0.5)**2 + (x[1]-0.5)**2) <= 0.2**2

    class Rectangle(SubDomain):
        def inside(self, x, on_boundary):
            return 0.3 <= x[0] <= 0.7 and 0.3 <= x[1] <= 0.7

    obstacle = Circle() if obstacle == 'circle' else Rectangle()
    subdomains = [None, obstacle]
    p_values = [1.0, 1E-4]

    boundary_conditions = {
        0: {'Neumann': 0},
        1: {'Neumann': 0},
        2: {'Dirichlet': Constant(1)}, # y=0
        3: {'Dirichlet': Constant(0)}, # y=1
        }

    f = Constant(0)
    Nx = Ny = 50
    u, p = solver_bc(
        p_values, f, boundary_conditions, Nx, Ny, degree=1,
        linear_solver='direct', subdomains=subdomains)
```

```
    v = flux(u, p)
    file = File('porous_media_flow.pvd')
    file << u
    file << v
    plot(u)
    plot(v)

if __name__ == '__main__':
    #application_test()
    #application_test_flux(Nx=20, Ny=20)
    #convergence_rate()
    #application_structured_mesh(2)
    #application_linalg()
    #application_bc_test_2mat()
    application_flow_around_circle()
    #test_solvers_bc()
    # Hold plot
    interactive()
```

Plotting the flux vector field is naturally as easy as plotting the gradient (see Section 1.2.6):

```
plot(flux, title='flux field')

flux_x, flux_y = flux.split(deepcopy=True)  # extract components
plot(flux_x, title='x-component of flux (-p*grad(u))')
plot(flux_y, title='y-component of flux (-p*grad(u))')
```

For data analysis of the nodal values of the flux field we can grab the underlying `numpy` arrays (demands a `deepcopy=True` in the split of `flux`):

```
flux_x_array = flux_x.vector().array()
flux_y_array = flux_y.vector().array()
```

The function `application_test_flux` in the program `ft06_poisson_vc.py` demonstrates the computations described above.

1.2.8 Creating the linear system explicitly

Given $a(u,v) = L(v)$, the discrete solution u is computed by inserting $u = \sum_{j=1}^{N} U_j \phi_j$ into $a(u,v)$ and demanding $a(u,v) = L(v)$ to be fulfilled for N test functions $\hat{\phi}_1, \ldots, \hat{\phi}_N$. This implies

$$\sum_{j=1}^{N} a(\phi_j, \hat{\phi}_i) U_j = L(\hat{\phi}_i), \quad i = 1, \ldots, N,$$

which is nothing but a linear system,

$$AU = b,$$

where the entries in A and b are given by

$$A_{ij} = a(\phi_j, \hat{\phi}_i),$$
$$b_i = L(\hat{\phi}_i)\,.$$

The examples so far have specified the left- and right-hand side of the variational formulation and then asked FEniCS to assemble the linear system and solve it. An alternative is to explicitly call functions for assembling the coefficient matrix A and the right-side vector b, and then solve the linear system $AU = b$ with respect to the U vector. Instead of `solve(a == L, u, b)` we now write

```
A = assemble(a)
b = assemble(L)
bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The variables `a` and `L` are as before. That is, `a` refers to the bilinear form involving a `TrialFunction` object (e.g., `u`) and a `TestFunction` object (`v`), and `L` involves a `TestFunction` object (`v`). From `a` and `L`, the `assemble` function can compute A and b.

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the call `bc.apply(A, b)` performs the necessary modifications of the linear system such that `u` is guaranteed to equal the prescribed boundary values. When we have multiple Dirichlet conditions stored in a list `bcs`, as explained in Section 1.4.2, we must apply each condition in `bcs` to the system:

```
# bcs is a list of DirichletBC objects
for bc in bcs:
    bc.apply(A, b)
```

There is an alternative function `assemble_system`, which can assemble the system and take boundary conditions into account in one call:

```
A, b = assemble_system(a, L, bcs)
```

The `assemble_system` function incorporates the boundary conditions in the element matrices and vectors, prior to assembly. The conditions are also incorporated in a symmetric way to preserve eventual symmetry of the coefficient matrix. With `bc.apply(A, b)` the matrix `A` is modified in an nonsymmetric way.

Note that the solution `u` is, as before, a `Function` object. The degrees of freedom, $U = A^{-1}b$, are filled into `u`'s `Vector` object (`u.vector()`) by the `solve` function.

The object `A` is of type `Matrix`, while `b` and `u.vector()` are of type `Vector`. We may convert the matrix and vector data to `numpy` arrays by calling the `array()` method as shown before. If you wonder how essential boundary conditions are incorporated in the linear system, you can print out `A` and `b` before and after the `bc.apply(A, b)` call:

```
A = assemble(a)
b = assemble(L)
if mesh.num_cells() < 16:  # print for small meshes only
    print(A.array())
    print(b.array())
bc.apply(A, b)
if mesh.num_cells() < 16:
    print(A.array())
    print(b.array())
```

With access to the elements in `A` through a `numpy` array we can easily perform computations on this matrix, such as computing the eigenvalues (using the `eig` function in `numpy.linalg`). We can alternatively dump `A.array()` and `b.array()` to file in MATLAB format and invoke MATLAB or Octave to analyze the linear system. Dumping the arrays to MATLAB format is done by

```
import scipy.io
scipy.io.savemat('Ab.mat', {'A': A.array(), 'b': b.array()})
```

Writing `load Ab.mat` in MATLAB or Octave will then make the array variables `A` and `b` available for computations.

Matrix processing in Python or MATLAB/Octave is only feasible for small PDE problems since the `numpy` arrays or matrices in MATLAB file format are dense matrices. FEniCS also has an interface to the eigensolver package SLEPc, which is a preferred tool for computing the eigenvalues of large, sparse matrices of the type encountered in PDE problems (see `demo/la/eigenvalue` in the FEniCS source code tree for a demo).

By default, `solve(A, U, b)` applies sparse LU decomposition as solver. Specification of an iterative solver and preconditioner is done through two optional arguments:

```
solve(A, U, b, 'cg', 'ilu')
```

Appropriate names of solvers and preconditioners are found in Section 1.2.2.

To control tolerances in the stopping criterion and the maximum number of iterations, one can explicitly form a `KrylovSolver` object and set items in its `parameters` attribute (see also Section 1.2.3):

```
solver = KrylovSolver('cg', 'ilu')
prm = solver.parameters
prm['absolute_tolerance'] = 1E-7
prm['relative_tolerance'] = 1E-4
prm['maximum_iterations'] = 1000
u = Function(V)
```

```
U = u.vector()
set_log_level(DEBUG)
solver.solve(A, U, b)
```

The function `solver_linalg` in the program file `ft06_poisson_vc.py` implements a solver function where the user can choose between different types of assembly: the variational (`solve(a == L, u, bc)`), assembling the matrix and right-hand side separately, and assembling the system such that the coefficient matrix preserves symmetry. The function `application_linalg` runs a test problem on sequence of meshes and solves the problem with symmetric and non-symmetric modification of the coefficient matrix. One can monitor the number of Krylov method iteration and realize that with a symmetric coefficient matrix, the Conjugate Gradient method requires slightly fewer iterations than GMRES in the non-symmetric case. Taking into account that the Conjugate Gradient method has less work per iteration, there is some efficiency to be gained by using `assemble_system`.

hpl 5: Running `application_linalg`, the results are strange: Why does the `solve(a==L,...)` method need many more iterations than `solve(A, U, b, ...)` when we use the same Krylov parameter settings? Something wrong with the settings?

The choice of start vector for the iterations in a linear solver is often important. With the `solver.solve(A, U, b)` call the default start vector is the zero vector. A start vector with random numbers in the interval $[-100, 100]$ can be computed as

```
n = u.vector().array().size
U = u.vector()
U[:] = numpy.random.uniform(-100, 100, n)
solver.parameters['nonzero_initial_guess'] = True
solver.solve(A, U, b)
```

Note that we must turn off the default behavior of setting the start vector ("initial guess") to zero, and then the provided value of `U` is used as start vector.

Creating the linear system explicitly in a program can have some advantages in more advanced problem settings. For example, A may be constant throughout a time-dependent simulation, so we can avoid recalculating A at every time level and save a significant amount of simulation time.

1.2.9 Taking advantage of structured mesh data

When finite element computations are done on a structured rectangular mesh, maybe with uniform partitioning, VTK-based tools for completely unstructured 2D/3D meshes are not required. Instead we can use visualization and data analysis tools for *structured data*. Such data typically appear in finite

difference simulations and image analysis. Analysis and visualization of structured data are faster and easier than doing the same with data on unstructured meshes, and the collection of tools to choose among is much larger. We shall demonstrate the potential of such tools and how they allow for tailored and flexible visualization and data analysis.

A necessary first step is to transform our `mesh` object to an object representing a rectangle with equally-shaped *rectangular* cells. The second step is to transform the one-dimensional array of nodal values to a two-dimensional array holding the values at the corners of the cells in the structured mesh. We want to access a value by its i and j indices, i counting cells in the x direction, and j counting cells in the y direction. This transformation is in principle straightforward, yet it frequently leads to obscure indexing errors, so using software tools to ease the work is advantageous.

In the directory `src/modules`, associated with this booklet, we have included a Python module `BoxField` that can take a finite element function `u` computed by a FEniCS software and represent it on a structured box-shaped mesh and assign or extract values by multi-dimensional indexing: `[i]` in 1D, `[i,j]` in 2D, and `[i,j,k]` in 3D. Given a finite element function `u`, the following function returns a `BoxField` object that represents `u` on a structured mesh:

```
def structured_mesh(u, divisions):
    """Represent u on a structured mesh."""
    # u must have P1 elements, otherwise interpolate to P1 elements
    u2 = u if u.ufl_element().degree() == 1 else \
         interpolate(u, FunctionSpace(mesh, 'P', 1))
    mesh = u.function_space().mesh()
    from BoxField import fenics_function2BoxField
    u_box = fenics_function2BoxField(
        u2, mesh, divisions, uniform_mesh=True)
    return u_box
```

Note that we can only turn functions on meshes with P1 elements into `BoxField` objects, so if `u` is based on another element type, we first interpolate the scalar field onto a mesh with P1 elements. Also note that to use the function, we need to know the divisions into cells in the various spatial directions (`divisions`).

The `u_box` object contains several useful data structures:

- `u_box.grid`: object for the structured mesh
- `u_box.grid.coor[X]`: grid coordinates in `X=0` direction
- `u_box.grid.coor[Y]`: grid coordinates in `Y=1` direction
- `u_box.grid.coor[Z]`: grid coordinates in `Z=2` direction
- `u_box.grid.coorv[X]`: vectorized version of `u_box.grid.coor[X]` (for vectorized computations or surface plotting)
- `u_box.grid.coorv[Y]`: vectorized version of `u_box.grid.coor[Y]`
- `u_box.grid.coorv[Z]`: vectorized version of `u_box.grid.coor[Z]`

- `u_box.values`: `numpy` array holding the `u` values; `u_box.values[i,j]` holds `u` at the mesh point with coordinates `(u_box.grid.coor[X], u_box.grid.coor[Y])`

Iterating over points and values. Let us go back to the `solver` function in the `ft06_poisson_vc.py` code from Section 1.2.7, compute `u`, map it onto a `BoxField` object for a structured mesh representation, and write out the coordinates and function values at all mesh points:

```
u = solver(p, f, u0, nx, ny, 1, linear_solver='direct')
u_box = structured_mesh(u, (nx, ny))
u_ = u_box.values       # numpy array
X = 0;  Y = 1           # for indexing in x and y direction

# Iterate over 2D mesh points (i,j)
print('u_ is defined on a structured mesh with %s points' %
      str(u_.shape))
for j in range(u_.shape[1]):
    for i in range(u_.shape[0]):
        print('u[%d,%d]=u(%g,%g)=%g' %
              (i, j,
               u_box.grid.coor[X][i], u_box.grid.coor[X][j],
               u_[i,j]))
```

Finite difference approximations. Note that with `u_`, we can easily express finite difference approximation of derivatives:

```
x = u_box.grid.coor[X]
dx = x[1] - x[0]
u_xx = (u_[i-1,j] - 2*u_[i,j] + u_[i+1,j])/dx**2
```

Surface plot. The ability to access a finite element field in the way one can access a finite difference-type of field is handy in many occasions, including visualization and data analysis. With Matplotlib we can create a surface plot, see Figure 1.2 (upper left):

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
fig = plt.figure()
ax = fig.gca(projection='3d')
cv = u_box.grid.coorv  # vectorized mesh coordinates
ax.plot_surface(cv[X], cv[Y], u_, cmap=cm.coolwarm,
                rstride=1, cstride=1)
plt.title('Surface plot of solution')
```

The key issue is to know that the coordinates needed for the surface plot is in `u_box.grid.coorv` and that the values are in `u_`.

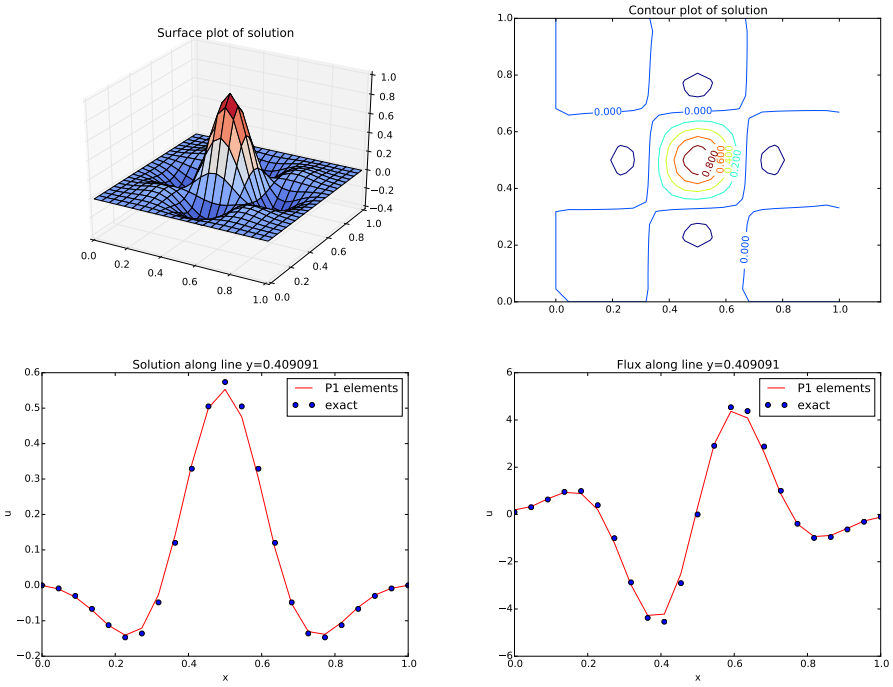


Fig. 1.2 Various plots of the solution on a structured mesh.

Contour plot. A contour plot can also be made by Matplotlib:

```
fig = plt.figure()
ax = fig.gca()
levels = [1.5, 2.0, 2.5, 3.5]
cs = ax.contour(cv[X], cv[Y], u_, levels=levels)
plt.clabel(cs)  # add labels to contour lines
plt.axis('equal')
plt.title('Contour plot of solution')
```

The result appears in Figure 1.2 (upper right).

Curve plot through the mesh. A handy feature of `BoxField` objects is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearest line of mesh points. In 3D fields one can also extract data in a plane. Say we want to plot u along the line $y = 0.4$. The mesh points, `x`, and the u values along this line, `u_val`, are extracted by

```
start = (0, 0.4)
X = 0
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line had to be snapped onto a gridline and in that case `y_fixed` holds the snapped (altered) y value. To avoid interpolation in the structured mesh, `snapped` is in fact *always* true.

A comparison of the numerical and exact solution along the line $y = 0.5$ (snapped from $y = 0.4$) is made by the following code:

```
start = (0, 0.4)
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
u_e_val = [u0((x_, y_fixed)) for x_ in x]

plt.figure()
plt.plot(x, u_val, 'r-')
plt.plot(x, u_e_val, 'bo')
plt.legend(['P1 elements', 'exact'], loc='upper left')
plt.title('Solution along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

See Figure 1.2 (lower left) for the resulting curve plot.

Curve plot of the flux. Let us also compare the numerical and exact flux $-p\partial u/\partial x$ along the same line as above:

```
flux_u = flux(u, p)
flux_u_x, flux_u_y = flux_u.split(deepcopy=True)

# Plot the numerical and exact flux along the same line
flux2_x = flux_u_x if flux_u_x.ufl_element().degree() == 1 \
          else interpolate(flux_x,
               FunctionSpace(u.function_space().mesh(),
                             'P', 1))
flux_u_x_box = structured_mesh(flux_u_x, (nx,ny))
x, flux_u_val, y_fixed, snapped = \
   flux_u_x_box.gridline(start, direction=X)
y = y_fixed

plt.figure()
plt.plot(x, flux_u_val, 'r-')
plt.plot(x, flux_u_x_exact(x, y_fixed), 'bo')
plt.legend(['P1 elements', 'exact'], loc='upper right')
plt.title('Flux along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

The second `plt.plot` command requires a Python function `flux_u_x_exact(x,y)` to be available for the exact flux expression.

Note that Matplotlib is one choice of plotting package. With the unified interface in the SciTools package[1] one can access Matplotlib, Gnuplot, MATLAB, OpenDX, VisIt, and other plotting engines through the same API.

Test problem. The graphics referred to in Figure 1.2 correspond to a test problem with prescribed solution $u_e = H(x)H(y)$, where

[1] `https://github.com/hplgit/scitools`

$$H(x) = e^{-16(x-\frac{1}{2})^2}\sin(3\pi x)\,.$$

We just fit a function $f(x,y)$ in the PDE (can choose $p=1$), and notice that $u=0$ along the boundary of the unit square. Although it is easy to carry out the differentiation of f by hand and hardcode the resulting expressions in an `Expression` object, a more reliable habit is to use Python's symbolic computing engine, SymPy, to perform mathematics and automatically turn formulas into C++ syntax for `Expression` objects. A short introduction was given in Section **??**.

We start out with defining the exact solution in `sympy`:

```
from sympy import exp, sin, pi  # for use in math formulas
import sympy as sym
H = lambda x: exp(-16*(x-0.5)**2)*sin(3*pi*x)
x, y = sym.symbols('x[0], x[1]')
u = H(x)*H(y)
```

Define symbolic coordinates as required in `Expression` objects

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

Turning the expression for `u` into C or C++ syntax for `Expression` objects needs two steps. First we ask for the C code of the expression,

```
u_c = sym.printing.ccode(u)
```

Printing out `u_c` gives (the output is here manually broken into two lines):

```
-exp(-16*pow(x[0] - 0.5, 2) - 16*pow(x[1] - 0.5, 2))*
sin(3*M_PI*x[0])*sin(3*M_PI*x[1])
```

The necessary syntax adjustment is replacing the symbol `M_PI` for π in C/C++ by `pi` (or `DOLFIN_PI`):

```
u_c = u_c.replace('M_PI', 'pi')
u0 = Expression(u_c)
```

Thereafter, we can progress with the computation of $f = -\nabla\cdot(p\nabla u)$:

```
p = 1
f = sym.diff(-p*sym.diff(u, x), x) + sym.diff(-p*sym.diff(u, y), y)
f = sym.simplify(f)
f_c = sym.printing.ccode(f)
f_c = f_c.replace('M_PI', 'pi')
f = Expression(f_c)
```

We also need a Python function for the exact flux $-p\partial u/\partial x$:

```
flux_u_x_exact = sym.lambdify([x, y], -p*sym.diff(u, x),
                              modules='numpy')
```

It remains to define `p = Constant(1)` and set `nx` and `ny` before calling `solver` to compute the finite element solution of this problem.

1.3 Postprocessing computations

hpl 3: Need a little intro.

1.3.1 Computing functionals

After the solution u of a PDE is computed, we occasionally want to compute functionals of u, for example,

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u\cdot\nabla u\,\mathrm{d}x, \tag{1.10}$$

which often reflects some energy quantity. Another frequently occurring functional is the error

$$||u_\mathrm{e}-u|| = \left(\int_\Omega (u_\mathrm{e}-u)^2\,\mathrm{d}x\right)^{1/2}, \tag{1.11}$$

where u_e is the exact solution. The error is of particular interest when studying convergence properties. Sometimes the interest concerns the flux out of a part Γ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma p\nabla u\cdot\boldsymbol{n}\,\mathrm{d}s, \tag{1.12}$$

where $\boldsymbol{n}$ is an outward unit normal at Γ and p is a coefficient (see the problem in Section 1.2.7 for a specific example). All these functionals are easy to compute with FEniCS, and this section describes how it can be done.

Energy functional. The integrand of the energy functional (1.10) is described in the UFL language in the same manner as we describe weak forms:

```
energy = 0.5*dot(grad(u), grad(u))*dx
E = assemble(energy)
```

The `assemble` call performs the integration. It is possible to restrict the integration to subdomains, or parts of the boundary, by using a mesh function to mark the subdomains as explained in Section 1.4.4.

Error functional. Computation of (1.11) is typically done by

```
error = (u - u_exact)**2*dx
E = sqrt(abs(assemble(error)))
```

The exact solution u_e is here in a `Function` or `Expression` object `u_exact`, while `u` is the finite element approximation. (Sometimes, for very small error values, the result of `assemble(error)` can be a (very small) negative number, so we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.)

As will be explained and demonstrate in Section 1.3.2, the integration of `(u - u_exact)**2*dx` can result in too optimistic convergence rates unless one is careful how `u_exact` is transferred onto a mesh. The general recommendation for reliable error computation is to use the `errornorm` function (see `pydoc fenics.errornorm` and Section 1.3.2 for more information):

```
E = errornorm(u_exact, u)
```

Flux Functionals. To compute flux integrals like $F = -\int_\Gamma p\nabla u \cdot \boldsymbol{n}\,\mathrm{d}s$ we need to define the $\boldsymbol{n}$ vector, referred to as *facet normal* in FEniCS. If the surface domain Γ in the flux integral is the complete boundary we can perform the flux computation by

```
n = FacetNormal(mesh)
flux = -p*dot(grad(u), n)*ds
total_flux = assemble(flux)
```

Although `grad(u)` and `grad(u)` are interchangeable in the above expression when `u` is a scalar function, we have chosen to write `grad(u)` because this is the right expression if we generalize the underlying equation to a vector Laplace/Poisson PDE. With `grad(u)` we must in that case write `dot(n, grad(u))`.

It is possible to restrict the integration to a part of the boundary using a mesh function to mark the relevant part, as explained in Section 1.4.4. Assuming that the part corresponds to subdomain number `i`, the relevant syntax for the variational formulation of the flux is `-p*dot(grad(u), n)*ds(i)`.

1.3.2 Computing convergence rates

hpl 7: Newer FEniCS examples have `dx(degree)`. Should explain that syntax. Also `Expression(string, degree)`.

To illustrate error computations and convergence of finite element solutions, we have included a function `convergence_rate` in the `ft06_poisson_vc.py` program. This is a tool that is very handy when verifying finite element codes and will therefore be explained in detail here.

The L^2 norm of the error in a finite element approximation u, u_e being the exact solution, is given by

Various ways of computing the error.

$$E = \left(\int_\Omega (u_e - u)^2 \, \mathrm{d}x \right)^{1/2},$$

and implemented in FEniCS by

```
error = (u - u_e)**2*dx
E = sqrt(abs(assemble(error)))
```

Sometimes, for very small error values, the result of `assemble(error)` can be a (very small) negative number, so we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.

We remark that `u_e` will, in the expression above, be interpolated onto the function space `V` before `assemble` can perform the integration over the domain. This implies that the exact solution used in the integral will vary linearly over the cells, and not as a sine function, if `V` corresponds to linear Lagrange elements. This situation may yield a smaller error `u - u_e` than what is actually true. More accurate representation of the exact solution is easily achieved by interpolating the formula onto a space defined by higher-order elements, say of third degree:

```
Ve = FunctionSpace(mesh, 'P', degree=3)
u_e_Ve = interpolate(u_e, Ve)
error = (u - u_e_Ve)**2*dx
E = sqrt(assemble(error))
```

To achieve complete mathematical control of which function space the computations are carried out in, we can explicitly interpolate `u` to the same space:

```
u_Ve = interpolate(u, Ve)
error = (u_Ve - u_e_Ve)**2*dx
```

The square in the expression for `error` will be expanded and lead to a lot of terms that almost cancel when the error is small, with the potential of introducing significant rounding errors. The function `errornorm` is available for avoiding this effect by first interpolating `u` and `u_exact` to a space with higher-order elements, then subtracting the degrees of freedom, and then performing the integration of the error field. The usage is simple:

```
E = errornorm(u_exact, u, normtype='L2', degree=3)
```

It is illustrative to look at the short implementation of `errornorm`:

```
def errornorm(u_exact, u, Ve):
    u_Ve = interpolate(u, Ve)
    u_e_Ve = interpolate(u_exact, Ve)
    e_Ve = Function(Ve)
    # Subtract degrees of freedom for the error field
    e_Ve.vector()[:] = u_e_Ve.vector().array() - \
                       u_Ve.vector().array()
    error = e_Ve**2*dx
```

```
    return sqrt(assemble(error))
```

The `errornorm` procedure turns out to be identical to computing the expression `(u_e - u)**2*dx` directly in the present test case.

Sometimes it is of interest to compute the error of the gradient field: $||\nabla(u-u_e)||$ (often referred to as the H^1 seminorm of the error). Given the error field `e_Ve` above, we simply write

```
H1seminorm = sqrt(assemble(dot(grad(e_Ve), grad(e_Ve))*dx))
```

All the various types of error computations here are placed in a function `compute_errors` in `ft06_poisson_vc.py`: **hpl 8**: Necessary to repeat code? New info is essentiall the return dict. **hpl 9**: Anders, I (in 2010...) ran into problems with `fenics.errornorm`, see comments in the code below, and made the version below. We should check out these problems again and adjust `fenics.errornorm` if necessary.

```
def compute_errors(u, u_exact):
    """Compute various measures of the error u - u_exact, where
    u is a finite element Function and u_exact is an Expression."""

    # Compute error norm (for very small errors, the value can be
    # negative so we run abs(assemble(error)) to avoid failure in sqrt

    V = u.function_space()

    # Function - Expression
    error = (u - u_exact)**2*dx
    E1 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_e onto the same space as u:
    u_e = interpolate(u_exact, V)
    error = (u - u_e)**2*dx
    E2 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_exact to higher-order elements,
    # u will also be interpolated to the space Ve before integration
    Ve = FunctionSpace(V.mesh(), 'P', 5)
    u_e = interpolate(u_exact, Ve)
    error = (u - u_e)**2*dx
    E3 = sqrt(abs(assemble(error)))

    # fenics.errornorm interpolates u and u_e to a space with
    # given degree, and creates the error field by subtracting
    # the degrees of freedom, then the error field is integrated
    # TEMPORARY BUG - doesn't accept Expression for u_e
    #E4 = errornorm(u_e, u, normtype='l2', degree=3)
    # Manual implementation errornorm to get around the bug:
    def errornorm(u_exact, u, Ve):
        u_Ve = interpolate(u, Ve)
        u_e_Ve = interpolate(u_exact, Ve)
        e_Ve = Function(Ve)
        # Subtract degrees of freedom for the error field
```

```
        e_Ve.vector()[:] = u_e_Ve.vector().array() - u_Ve.vector().array()
        # More efficient computation (avoids the rhs array result above)
        #e_Ve.assign(u_e_Ve)                      # e_Ve = u_e_Ve
        #e_Ve.vector().axpy(-1.0, u_Ve.vector())  # e_Ve += -1.0*u_Ve
        error = e_Ve**2*dx(Ve.mesh())
        return sqrt(abs(assemble(error))), e_Ve
    E4, e_Ve = errornorm(u_exact, u, Ve)

    # Infinity norm based on nodal values
    u_e = interpolate(u_exact, V)
    E5 = abs(u_e.vector().array() - u.vector().array()).max()

    # H1 seminorm
    error = dot(grad(e_Ve), grad(e_Ve))*dx
    E6 = sqrt(abs(assemble(error)))

    # Collect error measures in a dictionary with self-explanatory keys
    errors = {'u - u_exact': E1,
              'u - interpolate(u_exact,V)': E2,
              'interpolate(u,Ve) - interpolate(u_exact,Ve)': E3,
              'errornorm': E4,
              'infinity norm (of dofs)': E5,
              'grad(error) H1 seminorm': E6}

    return errors
```

Computing convergence rates empirically. Calling the `solver` function for finer and finer meshes enables us to study the convergence rate. Define the element size $h = 1/n$, where n is the number of cell divisions in x and y direction (`n=Nx=Ny` in the code). We perform experiments with $h_0 > h_1 > h_2 \cdots$ and compute the corresponding errors E_0, E_1, E_3 and so forth. Assuming $E_i = Ch_i^r$ for unknown constants C and r, we can compare two consecutive experiments, $E_i = Ch_i^r$ and $E_{i-1} = Ch_{i-1}^r$, and solve for r:

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})}.$$

The r values should approach the expected convergence rate `degree+1` as i increases.

The procedure above can easily be turned into Python code. Here we run through a different types of elements (P1, P2, P3, and P4), perform experiments over a series of refined meshes, and for each experiment report the six error types as returned by `compute_errors`:

```
def convergence_rate(u_exact, f, u0, p, degrees,
                     n=[2**(k+3) for k in range(5)]):
    """
    Compute convergence rates for various error norms for a
    sequence of meshes with Nx=Ny=b and P1, P2, ...,
    Pdegrees elements. Return rates for two consecutive meshes:
    rates[degree][error_type] = r0, r1, r2, ...
    """
```

```
    h = {}  # Discretization parameter, h[degree][experiment]
    E = {}  # Error measure(s), E[degree][experiment][error_type]
    P_degrees = 1,2,3,4
    num_meshes = 5

    # Perform experiments with meshes and element types
    for degree in P_degrees:
        n = 4   # Coarsest mesh division
        h[degree] = []
        E[degree] = []
        for i in range(num_meshes):
            n *= 2
            h[degree].append(1.0/n)
            u = solver(p, f, u0, n, n, degree,
                       linear_solver='direct')
            errors = compute_errors(u, u_exact)
            E[degree].append(errors)
            print('2*(%dx%d) P%d mesh, %d unknowns, E1=%g' %
                  (n, n, degree, u.function_space().dim(),
                   errors['u - u_exact']))
    # Convergence rates
    from math import log as ln  # log is a fenics name too
    error_types = list(E[1][0].keys())
    rates = {}
    for degree in P_degrees:
        rates[degree] = {}
        for error_type in sorted(error_types):
            rates[degree][error_type] = []
            for i in range(num_meshes):
                Ei   = E[degree][i][error_type]
                Eim1 = E[degree][i-1][error_type]
                r = ln(Ei/Eim1)/ln(h[degree][i]/h[degree][i-1])
                rates[degree][error_type].append(round(r,2))
    return rates

def convergence_rate_sin():
    """Compute convergence rates for u=sin(x)*sin(y) solution."""
    omega = 1.0
    u_exact = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                         omega=omega)
    f = 2*omega**2*pi**2*u_exact
    u0 = Constant(0)
    p = Constant(1)
    # Note: P4 for n>=128 seems to break down
    rates = convergence_rates(u_exact, f, u0, p, degrees=4,
                              n=[2**(k+3) for k in range(5)])
    # Print rates
    print('\n\n')
    for error_type in error_types:
        print(error_type)
        for degree in P_degrees:
            print('P%d: %s' %
                  (degree, str(rates[degree][error_type])[1:-1]))
```

Note how we make a complete general function `convergence_rate`, aimed at any 2D Poisson problem in the class we now can solve, and then call this general function in `convergence_rate_sin` for a special test case.

Test problem. Section 1.2.6 specifies a more complicated solution,

$$u(x,y) = \sin(\omega\pi x)\sin(\omega\pi y)$$

on the unit square. This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With ω restricted to an integer it follows that $u_0 = 0$.

We need to define the appropriate boundary conditions, the exact solution, and the f function in the code:

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

bc = DirichletBC(V, Constant(0.0), boundary)

omega = 1.0
u_e = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                 omega=omega)

f = 2*pi**2*omega**2*u_e
```

Experiments. Calling `convergence_rate_sin()` gives some interesting results. Using the error measure `E5` based on the infinity norm of the difference of the degrees of freedom, we have

element	$n=8$	$n=16$	$n=32$	$n=64$	$n=128$
P1	1.99	1.97	1.99	2.0	2.0
P2	3.99	3.96	3.99	4.0	3.99
P3	3.96	3.89	3.96	3.99	4.0
P4	3.75	4.99	5.0	5.0	

The computations with P4 elements on a 128×128 with a direct solver (UMFPACK) on a small laptop broke down. Otherwise we achieve expected results: the error goes like h^{d+1} for elements of degree d. Also L^2 norms based on the `errornorm` gives the expected h^{d+1} rate for u and h^d for ∇u.

However, using `(u - u_exact)**2` for the error computation, which implies interpolating `u_exact` onto the same space as `u`, results in h^4 convergence for P2 elements.

element	$n=8$	$n=16$	$n=32$	$n=64$	$n=128$
P1	1.98	1.94	1.98	2.0	2.0
P2	3.98	3.95	3.99	3.99	3.99
P3	3.69	4.03	4.01	3.95	2.77

This is an example where it is important to interpolate `u_exact` to a higher-order space (polynomials of degree 3 are sufficient here) to avoid computing a too optimistic convergence rate.

Checking convergence rates is the next best method for verifying PDE codes (the best being a numerical solution without approximation errors as in Section 1.2.4 and many other places in this tutorial).

1.4 Multiple domains and boundaries

hpl 3: Need a little intro.

1.4.1 Combining Dirichlet and Neumann conditions

Let us make a slight extension of our two-dimensional Poisson problem from Chapter **??** and add a Neumann boundary condition. The domain is still the unit square, but now we set the Dirichlet condition $u = u_0$ at the left and right sides, $x = 0$ and $x = 1$, while the Neumann condition

$$-\frac{\partial u}{\partial n} = g$$

is applied to the remaining sides $y = 0$ and $y = 1$. The Neumann condition is also known as a *natural boundary condition* (in contrast to an essential boundary condition).

PDE problem. Let Γ_D and Γ_N denote the parts of $\partial\Omega$ where the Dirichlet and Neumann conditions apply, respectively. The complete boundary-value problem can be written as

$$-\nabla^2 u = f \text{ in } \Omega, \tag{1.13}$$

$$u = u_0 \text{ on } \Gamma_D, \tag{1.14}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N\,. \tag{1.15}$$

Again we choose $u = 1 + x^2 + 2y^2$ as the exact solution and adjust f, g, and u_0 accordingly:

$$f = -6,$$
$$g = \begin{cases} -4, & y = 1 \\ 0, & y = 0 \end{cases}$$
$$u_0 = 1 + x^2 + 2y^2 .$$

For ease of programming we may introduce a g function defined over the whole of Ω such that g takes on the right values at $y = 0$ and $y = 1$. One possible extension is

$$g(x, y) = -4y .$$

Variational formulation. The first task is to derive the variational problem. This time we cannot omit the boundary term arising from the integration by parts, because v is only zero on Γ_D. We have

$$-\int_\Omega (\nabla^2 u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s,$$

and since $v = 0$ on Γ_D,

$$-\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s = -\int_{\Gamma_N} \frac{\partial u}{\partial n} v \,\mathrm{d}s = \int_{\Gamma_N} g v \,\mathrm{d}s,$$

by applying the boundary condition on Γ_N. The resulting weak form reads

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_N} g v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x . \tag{1.16}$$

Expressing this equation in the standard notation $a(u, v) = L(v)$ is straightforward with

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.17}$$
$$L(v) = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_N} g v \,\mathrm{d}s . \tag{1.18}$$

Implementation. How does the Neumann condition impact the implementation? Let us go back to the very simplest file, `ft01_poisson_flat.py`, from Section **??**, we realize that the statements remain almost the same. Only two adjustments are necessary:

- The function describing the boundary where Dirichlet conditions apply must be modified.
- The new boundary term must be added to the expression in `L`.

The first adjustment can be coded as

```
def Dirichlet_boundary(x, on_boundary):
```

```
    if on_boundary:
        if x[0] == 0 or x[0] == 1:
            return True
        else:
            return False
    else:
        return False
```

A more compact implementation reads

```
def Dirichlet_boundary(x, on_boundary):
    return on_boundary and (x[0] == 0 or x[0] == 1)
```

Never use == for comparing real numbers!

A list like `x[0] == 1` should never be used if `x[0]` is a real number, because rounding errors in `x[0]` may make the test fail even when it is mathematically correct. Consider

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Comparison of real numbers need to use tolerances! The values of the tolerances depend on the size of the numbers involved in arithmetic operations:

```
>>> abs(0.1+0.2 - 0.3)
5.551115123125783e-17
>>> abs(1.1+1.2 - 2.3)
0.0
>>> abs(10.1+10.2 - 20.3)
3.552713678800501e-15
>>> abs(100.1+100.2 - 200.3)
0.0
>>> abs(1000.1+1000.2 - 2000.3)
2.2737367544323206e-13
>>> abs(10000.1+10000.2 - 20000.3)
3.637978807091713e-12
```

For numbers around unity, tolerances as low as $3 \cdot 10^{-16}$ can be used (in fact, this tolerance is known as the constant `DOLFIN_EPS` in FEniCS), otherwise an appropriate tolerance must be found.

Testing for `x[0] == 1` should therefore be implemented as

```
tol = 1E-14
if abs(x[0] - 1) < tol:
    ...
```

Here is a new boundary function using tolerances in the test:

```
def Dirichlet_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and \
           (abs(x[0]) < tol or abs(x[0] - 1) < tol)
```

The second adjustment of our program concerns the definition of `L`, where we have to add a boundary integral and a definition of the g function to be integrated:

```
g = Expression('-4*x[1]')
L = f*v*dx - g*v*ds
```

The `ds` variable implies a boundary integral, while `dx` implies an integral over the domain Ω. No more modifications are necessary.

1.4.2 Multiple Dirichlet conditions

The PDE problem from the previous section applies a function $u_0(x,y)$ for setting Dirichlet conditions at two parts of the boundary. Having a single function to set multiple Dirichlet conditions is seldom possible. The more general case is to have m functions for setting Dirichlet conditions on m parts of the boundary. The purpose of this section is to explain how such multiple conditions are treated in FEniCS programs.

Let us return to the case from Section 1.4.1 and define two separate functions for the two Dirichlet conditions:

$$\begin{aligned} -\nabla^2 u &= -6 \text{ in } \Omega, \\ u &= u_L \text{ on } \Gamma_0, \\ u &= u_R \text{ on } \Gamma_1, \\ -\frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N . \end{aligned}$$

Here, Γ_0 is the boundary $x=0$, while Γ_1 corresponds to the boundary $x=1$. We have that $u_L = 1+2y^2$, $u_R = 2+2y^2$, and $g=-4y$.

Functions for marking Dirichlet boundaries. For the left boundary Γ_0 we define the usual triple of a function for the boundary value, a function for defining the boundary of interest, and a `DirichletBC` object:

```
u_L = Expression('1 + 2*x[1]*x[1]')

def left_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0]) < tol

Gamma_0 = DirichletBC(V, u_L, left_boundary)
```

For the boundary $x = 1$ we write a similar code snippet:

```
u_R = Expression('2 + 2*x[1]*x[1]')

def right_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0] - 1) < tol

Gamma_1 = DirichletBC(V, u_R, right_boundary)
```

The various essential conditions are then collected in a list and used in the solution process:

```
bcs = [Gamma_0, Gamma_1]
...
solve(a == L, u, bcs)
# or
problem = LinearVariationalProblem(a, L, u, bcs)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

In other problems, where the u values are constant at a part of the boundary, we may use a simple `Constant` object instead of an `Expression` object.

1.4.3 Working with subdomains

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, these kind of problems are handled by defining subdomains inside the domain. The subdomains may represent the various materials. We can thereafter define material properties through functions, known in FEniCS as *mesh functions*, that are piecewise constant in each subdomain. A simple example with two materials (subdomains) in 2D will demonstrate the basic steps in the process.

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{1.19}$$

in a domain Ω consisting of two subdomains where k takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for the current implementation the domain $\Omega = [0,1]\times[0,1]$ and divide it into two equal subdomains, as depicted in Figure 1.3,

$$\Omega_0 = [0,1]\times[0,1/2], \quad \Omega_1 = [0,1]\times(1/2,1]\,.$$

We define $k(x,y) = k_0$ in Ω_0 and $k(x,y) = k_1$ in Ω_1, where $k_0 > 0$ and $k_1 > 0$ are given constants.

Physically, the present problem may correspond to heat conduction, where the heat conduction in Ω_1 is more efficient than in Ω_0. An alternative inter-

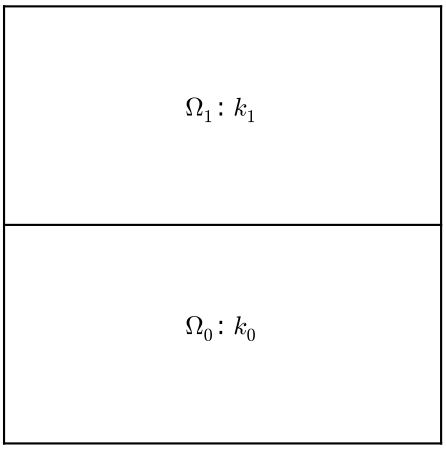


Fig. 1.3 Medium with discontinuous material properties.

pretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differ.

Expression objects with if test. The simplest way of implementing a variable k is to define an `Expression` object where we return the appropriate k value depending on the position in space. Since we need some testing on the coordinates, the most straightforward approach is to define a subclass of `Expression`, where we can use a full Python method instead of just a C++ string formula for specifying a function. The method that defines the function is called `eval`:

```
class K(Expression):
    def set_k_values(self, k0, k1):
        self.k0, self.k1 = k0, k1

    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
    # Fill in-place value[0] for scalar function,
    # value[:] for vector function (no return)

    tol = 1E-14  # Tolerance for coordinate comparisons
        if x[1] <= 0.5+tol:
        value[0] = self.k0
    else:
        value[0] = self.k1

# Initialize
k = K()
k.set_k_values(1, 0.01)
```

The `eval` method gives great flexibility in defining functions, but a downside is that C++ calls up `eval` in Python for each point `x`, which is a slow process,

and the number of calls is proportional to the number of numerical integration points in the mesh (about the number of degrees of freedom). Function expressions in terms of strings are compiled to efficient C++ functions, being called from C++, so we should try to express functions as string expressions if possible. (The `eval` method can also be defined through C++ code, but this is much more complicated and not covered here.) The idea is to use inline if tests in C++:

```
tol = 1E-14
k0 = 1.0
k1 = 0.01
k = Expression('x[1] <= 0.5+tol? k0 : k1',
               tol=tol, k0=k0, k1=k1)
```

The method with if tests on the location is feasible when the subdomains have very simple shapes. A completely general method, utilizing *mesh functions*, is described next.

Mesh functions. We now address how to specify the subdomains Ω_0 and Ω_1 so that the method also works for subdomains of any shape. For this purpose we need to use subclasses of class `SubDomain`, not only plain functions as we have used so far for specifying boundaries. Consider the boundary function

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

for defining the boundary $x = 0$. Instead of using such a stand-alone function, we can create an instance (or object) of a subclass of `SubDomain`, which implements the `inside` method as an alternative to the `boundary` function:

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0]) < tol

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

A word about computer science terminology may be used here: The term *instance* means a Python object of a particular type (such as `SubDomain`, `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangeable terms. In other computer programming languages one may also use the term *variable* for the same thing. We mostly use the well-known term *object* in this text.

A subclass of `SubDomain` with an `inside` method offers functionality for marking parts of the domain or the boundary. Now we need to define one class for the subdomain Ω_0 where $y \leq 1/2$ and another for the subdomain Ω_1 where $y \geq 1/2$:

```
tol = 1E-14  # Tolerance for coordinate comparisons
```

```
class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] <= 0.5+tol

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] >= 0.5-tol
```

Notice the use of `<=` and `>=` in both tests. For a cell to belong to, e.g., Ω_1, the `inside` method must return `True` for all the vertices `x` of the cell. So to make the cells at the internal boundary $y = 1/2$ belong to Ω_1, we need the test `x[1] >= 0.5`. However, because of potential rounding errors in the coordinates `x[1]`, we use a tolerance in the comparisons: `x[1] >= 0.5-tol`.

The next task is to use a *mesh function* to mark all cells in Ω_0 with the subdomain number 0 and all cells in Ω_1 with the subdomain number 1. Our convention is to number subdomains as $0, 1, 2, \ldots$.

A `MeshFunction` object is a discrete function that can be evaluated at a set of so-called *mesh entities*. Examples of mesh entities are cells, facets, and vertices. A `MeshFunction` over cells is suitable to represent subdomains (materials), while a `MeshFunction` over facets is used to represent pieces of external or internal boundaries. Mesh functions over vertices can be used to describe continuous fields. The specialized classes `CellFunction` and `FacetFunction` are used to construct mesh functions of cells and facets, respectively.

Since we need to define subdomains of Ω in the present example, we make use of a `CellFunction`. The constructor is fed with two arguments: 1) the type of value: `'int'` for integers, `'uint'` for positive (unsigned) integers, `'double'` for real numbers, and `'bool'` for logical values; 2) a `Mesh` object. Alternatively, the constructor can take just a filename and initialize the `CellFunction` from data in a file.

We start with creating a `CellFunction` whose values are non-negative integers (`'uint'`) for numbering the subdomains. The appropriate code for two subdomains then reads

```
materials = CellFunction('size_t', mesh)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(materials, 0)
subdomain1 = Omega1()
subdomain1.mark(materials, 1)

# Alternative
materials.set_all(0)
subdomain1.mark(materials, 1)
```

Calling `materials.array()` returns a `numpy` array of the subdomain values. That is, `materials.array()[i]` is the subdomain value of cell number `i`. This array is used to look up the subdomain or material number of a specific element.

We need a function `k` that is constant in each subdomain Ω_0 and Ω_1. Since we want `k` to be a finite element function, it is natural to choose a space of functions that is constant over each element. The family of discontinuous Galerkin methods, in FEniCS denoted by `'DG'`, is suitable for this purpose. Since we want functions that are piecewise constant, the value of the degree parameter is zero:

```
V0 = FunctionSpace(mesh, 'DG', 0)
k  = Function(V0)
```

To fill `k` with the right values in each element, we loop over all cells (i.e., indices in `materials.array()`), extract the corresponding subdomain number of a cell, and assign the corresponding k value to the `k.vector()` array:

```
k_values = [1.5, 50]  # values of k in the two subdomains
for cell_no in range(len(materials.array())):
    material_no = materials.array()[cell_no]
    k.vector()[cell_no] = k_values[material_no]
```

Long loops in Python are known to be slow, so for large meshes it is preferable to avoid such loops and instead use *vectorized code*. Normally this implies that the loop must be replaced by calls to functions from the `numpy` library that operate on complete arrays (in efficient C code). The functionality we want in the present case is to compute an array of the same size as `materials.array()`, but where the value `i` of an entry in `materials.array()` is replaced by `k_values[i]`. Such an operation is carried out by the `numpy` function `choose`:

```
help = numpy.asarray(materials.array(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The `help` array is required since `choose` cannot work with `materials.array()` because this array has elements of type `uint32`. We must therefore transform this array to an array `help` with standard `int32` integers.

The next section exemplifies a complete solver with a piecewise constant coefficient, like k, defined through `SubDomain` objects, combined with different types of boundary conditions.

C++ strings for subdomain definitions. The `SubDomain` class in Python is convenient, but leads to lots of function calls from C++ to Python, which are slow. In large problems, the subdomains should be defined through C++ code. This is easy to achieve using the `CompiledSubDomain` object. Consider the definition of classes `Omega0` and `Omega1` above in Python. The key strings that define these subdomain can be expressed in C++ syntax and fed to `CompiledSubDomain` as follows:

```
tol = 1E-14  # Tolerance for coordinate comparisons

subdomain0 = CompiledSubDomain(
                'x[1] <= boundary+tol', tol=1E-14, boundary=0.5)
subdomain1 = CompiledSubDomain(
```

```
                'x[1] >= boundary-tol', tol=1E-14, boundary=0.5)
```

As seen, one can have parameters in the strings and specify their values by keyword arguments. The resulting objects, `subdomain0` and `subdomain1`, can be used as ordinary `SubDomain` objects.

Compiled subdomain strings can be applied for specifying boundaries as well, e.g.,

```
y_R = CompiledSubDomain('on_boundary && near(x[1], R, eps=tol)',
                        tol=1E-14, R=2)   # y=2
```

It is possible to feed the C++ string (without parameters) directly as the third argument to `DirichletBC` without explicitly constructing a `CompiledSubDomain` object:

```
bc1 = DirichletBC(V, value, 'on_boundary && near(x[1], 2, 1E-14)')
```

1.4.4 Multiple Neumann, Robin, and Dirichlet condition

Consider the model problem from Section 1.4.2 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

Three types of boundary conditions. We extend our repertoire of boundary conditions to three types: Dirichlet, Neumann, and Robin. Dirichlet conditions apply to some parts $\Gamma_{D,0}$, $\Gamma_{D,1}$, ..., of the boundary:

$$u_{0,0} \text{ on } \Gamma_{D,0}, \quad u_{0,1} \text{ on } \Gamma_{D,1}, \ldots$$

where $u_{0,i}$ are prescribed functions, $i = 0, 1, \ldots$ On other parts, $\Gamma_{N,0}$, $\Gamma_{N,1}$, and so on, we have Neumann conditions

$$-p\frac{\partial u}{\partial n} = g_0 \text{ on } \Gamma_{N,0}, \quad -p\frac{\partial u}{\partial n} = g_1 \text{ on } \Gamma_{N,1}, \quad \ldots$$

Finally, we have *Robin conditions*

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

where r and s are specified functions. The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law. In that case, r is a heat transfer coefficient, and s is the temperature of the surroundings. Both can be space and time-dependent. The Robin conditions apply at some parts $\Gamma_{R,0}$, $\Gamma_{R,1}$, and so forth:

$$-p\frac{\partial u}{\partial n} = r_0(u-s_0) \text{ on } \Gamma_{R,0}, \quad -p\frac{\partial u}{\partial n} = r_1(u-s_1) \text{ on } \Gamma_{R,1}, \quad \ldots$$

A general model problem. With the notation above, the model problem to be solved with multiple Dirichlet, Neumann, and Robin conditions can formally be defined as

$$-\nabla\cdot(p\nabla u) = -f, \text{ in } \Omega, \tag{1.20}$$

$$u = u_{0,i} \text{ on } \Gamma_{D,i}, \quad i = 0,1,\ldots \tag{1.21}$$

$$-p\frac{\partial u}{\partial n} = g_i \text{ on } \Gamma_{N,i}, \quad i = 0,1,\ldots \tag{1.22}$$

$$-p\frac{\partial u}{\partial n} = r_i(u-s_i) \text{ on } \Gamma_{R,i}, \quad i = 0,1,\ldots \tag{1.23}$$

Variational formulation. Integration by parts of $-\int_\Omega v\nabla\cdot(p\nabla u)\,\mathrm{d}x$ becomes as usual

$$-\int_\Omega v\nabla\cdot(p\nabla u)\,\mathrm{d}x = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s\,.$$

The boundary integral does not apply to the parts of the boundary where we have Dirichlet conditions ($\Gamma_{D,i}$). Moreover, on the remaining parts, we must split the boundary integral into the parts where we have Neumann and Robin conditions such that we insert the right conditions as integrands. Specifically, we have

$$\begin{aligned}-\int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s &= -\sum_i\int_{\Gamma_{N,i}} p\frac{\partial u}{\partial n}\,\mathrm{d}s - \sum_i\int_{\Gamma_{R,i}} p\frac{\partial u}{\partial n}\,\mathrm{d}s \\ &= \sum_i\int_{\Gamma_{N,i}} g_i\,\mathrm{d}s + \sum_i\int_{\Gamma_{R,i}} r_i(u-s_i)\,\mathrm{d}s\,.\end{aligned}$$

The variational formulation then becomes

$$F = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i \int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i \int_{\Gamma_{R,i}} r_i(u-s_i)v\,\mathrm{d}s - \int_\Omega fv\,\mathrm{d}x = 0\,. \tag{1.24}$$

We have been used to writing this variational formulation in the standard notation $a(u,v) = L(v)$, which requires that we identify all integrals with *both* u and v, and collect these in $a(u,v)$, while the remaining integrals with v and not u go into $L(v)$. The integral from the Robin condition must of this reason be split in two parts:

$$\int_{\Gamma_{R,i}} r_i(u-s_i)v\,\mathrm{d}s = \int_{\Gamma_{R,i}} r_i uv\,\mathrm{d}s - \int_{\Gamma_{R,i}} r_i s_i v\,\mathrm{d}s\,.$$

We then have

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i \int_{\Gamma_{R,i}} r_i uv\,\mathrm{d}s, \tag{1.25}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x - \sum_i \int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i \int_{\Gamma_{R,i}} r_i s_i v\,\mathrm{d}s\,. \tag{1.26}$$

Implementation of boundary conditions. Looking at our previous `solver` functions for solving the 2D Poisson equation, the following new aspects must be taken care of:

1. definition of a mesh function over the boundary,
2. marking each side as a subdomain, using the mesh function,
3. splitting a boundary integral into parts.

A general approach to the first task is to mark each of the desired boundaries with markers 0, 1, 2, and so forth. Here we aim at the four sides of the unit square, marked with 0 ($x = 0$), 1 ($x = 1$), 2 ($y = 0$), and 3 ($y = 1$). The marking of boundaries makes use of a mesh function object, but contrary to Section 1.4.3, this is not a function over cells, but a function over cell facets. We apply the `FacetFunction` for this purpose:

```
boundary_parts = FacetFunction('size_t', mesh)
```

As in Section 1.4.3 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated geometries may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $x = 0$ boundary can be marked by

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

bx0 = BoundaryX0()
bx0.mark(boundary_parts, 0)
```

Similarly, we make the classes `BoundaryX1` for the $x = 1$ boundary, `BoundaryY0` for the $y = 0$ boundary, and `BoundaryY1` for the $y = 1$ boundary, and mark these as subdomains 1, 2, and 3, respectively.

For generality of the implementation, we let the user specify what kind of boundary condition that applies to each of the four boundaries. We set up a Python dictionary for this purpose, with the key as subdomain number and the value as a dictionary specifying the kind of condition as key and a function as its value. For example,

```
boundary_conditions = {
  0: {'Dirichlet': u0},
  1: {'Robin': (r, s)},
  2: {'Neumann: g}},
  3: {'Neumann', 0}}
```

specifies

- a Dirichlet condition, with values implemented by an `Expression` or `Constant` object `u0`, on subdomain 0, i.e., the $x = 1$ boundary;
- a Robin condition (1.4.4) on subdomain 1, $x = 1$, with `Expression` or `Constant` objects `r` and `s` specifying r and s;
- a Neumann condition $\partial u/\partial n = g$ on subdomain 2, $y = 0$, where an `Expression` or `Constant` object `g` implements the value g;
- a homogeneous Neumann condition $\partial u/\partial n = 0$ on subdomain 3, $y = 1$.

As explained in Section 1.4.2, multiple Dirichlet conditions must be collected in a list of `DirichletBC` objects. Based on the `boundary_conditions` data structure above, we can construct this list by the following snippet:

```
bcs = []  # List of Dirichlet conditions
for n in boundary_conditions:
    if 'Dirichlet' in boundary_conditions[n]:
        bcs.append(
            DirichletBC(V, boundary_conditions[n]['Dirichlet'],
                        boundary_parts, n))
```

The new aspect of the variational problem is the two distinct boundary integrals over $\Gamma_{N,i}$ and $\Gamma_{R,i}$. Having a mesh function over exterior cell facets (our `boundary_parts` object), where subdomains (boundary parts) are numbered as $0, 1, 2, \ldots$, the special symbol `ds(0)` implies integration over subdomain (part) 0, `ds(1)` denotes integration over subdomain (part) 1, and so on. The idea of multiple `ds`-type objects generalizes to volume integrals too: `dx(0)`, `dx(1)`, etc., are used to integrate over subdomain 0, 1, etc., inside Ω.

Before we have `ds(n)` for integers `n` defined, we must do

```
ds = Measure('ds', domain=mesh, subdomain_data=boundaries_parts)
```

Similarly, if we want integration of different parts of the domain, we redefine `dx` as

```
dx = Measure('dx', domain=mesh, subdomain_data=domains)
```

where `domains` is a `CellFunction` defining subdomains in Ω.

Suppose we have a Robin condition with values `r` and `s` on subdomain `R`, a Neumann condition with value `g` on subdomain `N`, the variational form can be written

```
a = dot(grad(u), grad(v))*dx + r*u*v*ds(R)
L = f*v*dx - g*v*ds(N) + r*s*v*ds(R)
```

In our case things get a bit more complicated since the information about integrals in Neumann and Robin conditions are in the `boundary_conditions` data structure. We can collect all Neumann conditions by the code

```
u = TrialFunction(V)
v = TestFunction(V)
Neumann_integrals = []
for n in boundary_conditions:
    if 'Neumann' in boundary_conditions[n]:
        if boundary_conditions[n]['Neumann'] != 0:
            g = boundary_conditions[n]['Neumann']
            Neumann_integrals.append(g*v*ds(n))
```

Applying `sum(Nemann_integrals)` will apply the `+` operator to the variational forms in the `Numeann_integrals` list and result in the integrals we need for the right-hand side `L` of the variational form.

The integrals in the Robin condition can similarly be collected in lists:

```
Robin_a_integrals = []
Robin_L_integrals = []
for n in boundary_conditions:
    if 'Robin' in boundary_conditions[n]:
        r, s = boundary_conditions[n]['Robin']
        Robin_a_integrals.append(r*u*v*ds(n))
        Robin_L_integrals.append(r*s*v*ds(n))
```

We are now in a position to define the `a` and `L` expressions in the variational formulation:

```
a = dot(p*grad(u), grad(v))*dx + \
    sum(Robin_a_integrals)
L = f*v*dx - sum(Neumann_integrals) + sum(Robin_L_integrals)
```

Simplified handling of the variational formulation. We carefully ordered the terms in the variational formulation above into the a and L parts. This requires a splitting of the Robin condition and makes the `a` and `L` expressions less readable (still we think understanding this splitting is key for any finite element programmer!). Fortunately, UFL allow us to specify the complete variational form (1.24) as *one expression* and offer tools to extract what goes into the bilinear form $a(u,v)$ and the linear form $L(v)$:

```
F = dot(p*grad(u), grad(v))*dx + \
    sum(Robin_integrals) - f*v*dx + sum(Neumann_integrals)
a, L = lhs(F), rhs(F)
```

This time we can more naturally define the integrals from the Robin condition as `r*(u-s)*v*ds(n)`:

```
Robin_integrals = []
for n in boundary_conditions:
    if 'Robin' in boundary_conditions[n]:
        r, s = boundary_conditions[n]['Robin']
        Robin_integrals.append(r*(u-s)*v*ds(n))
```

The complete code is in the `solver_bc` function in the `ft06_poisson_vc.py` file.

Test problem. Let us continue to use $u_{\mathrm{e}} = 1 + x^2 + 2y^2$ as the exact solution, and set $p = 1$ and $f = -6$ in the PDE. Our domain is the unit square, and we assign Dirichlet conditions at $x = 0$ and $x = 1$, a Neumann condition at $y = 1$, and a Robin condition at $y = 0$. With the given u_{e}, we realize that the Neumann condition is $-4y$ (which means -4 at $y = 1$), while the Robin condition can be selected in many ways. Since $\partial u/\partial n = -\partial u/\partial y = 0$ at $y = 0$, we can select $s = u$ and have r arbitrary in the Robin condition.

The boundary parts are $\Gamma_{D,0}$: $x = 0$, $\Gamma_{D,1}$: $x = 1$, $\Gamma_{R,0}$: $y = 0$, and $\Gamma_{N,0}$: $y = 1$.

When implementing this test problem (and especially other test problems with more complicated expressions), it is advantageous to use symbolic computing. Below we define u_{e} as a `sympy` expression and derive other functions from their mathematical definitions. Then we turn these expressions into C/C++ code, which can be fed into `Expression` objects.

```
def application_bc_test():
    # Define manufactured solution in sympy and derive f, g, etc.
    import sympy as sym
    x, y = sym.symbols('x[0] x[1]')  # UFL needs x[0] for x etc.
    u = 1 + x**2 + 2*y**2
    f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)  # -Laplace(u)
    f = sym.simplify(f)
    u_00 = u.subs(x, 0)  # x=0 boundary
    u_01 = u.subs(x, 1)  # x=1 boundary
    g = -sym.diff(u, y).subs(y, 1)  # x=1 boundary, du/dn=-du/dy
    r = 1000 # any function can go here
    s = u

    # Turn to C/C++ code for UFL expressions
    f = sym.printing.ccode(f)
    u_00 = sym.printing.ccode(u_00)
    u_01 = sym.printing.ccode(u_01)
    g = sym.printing.ccode(g)
    r = sym.printing.ccode(r)
    s = sym.printing.ccode(s)
    print('Test problem (C/C++):\nu = %s\nf = %s' % (u, f))
    print('u_00: %s\nu_01: %s\ng = %s\nr = %s\ns = %s' %
          (u_00, u_01, g, r, s))

    # Turn into FEniCS objects
```

```
    u_00 = Expression(u_00)
    u_01 = Expression(u_01)
    f = Expression(f)
    g = Expression(g)
    r = Expression(r)
    s = Expression(s)
    u_exact = Expression(sym.printing.ccode(u))

    boundary_conditions = {
        0: {'Dirichlet': u_00},   # x=0
        1: {'Dirichlet': u_01},   # x=1
        2: {'Robin': (r, s)},     # y=0
        3: {'Neumann': g}}        # y=1

    p = Constant(1)
    Nx = Ny = 2
    u, p = solver_bc(
        p, f, boundary_conditions, Nx, Ny, degree=1,
        linear_solver='direct',
        debug=2*Nx*Ny < 50,  # for small problems only
        )

    # Compute max error in infinity norm
    u_e = interpolate(u_exact, u.function_space())
    import numpy as np
    max_error = np.abs(u_e.vector().array() -
                       u.vector().array()).max()
    print('Max error:', max_error)

    # Print numerical and exact solution at the vertices
    if u.function_space().dim() < 50:  # (small problems only)
        u_e_at_vertices = u_e.compute_vertex_values()
        u_at_vertices = u.compute_vertex_values()
        coor = u.function_space().mesh().coordinates()
        for i, x in enumerate(coor):
            print('vertex %2d (%9g,%9g): error=%g %g vs %g'
                  % (i, x[0], x[1],
                     u_e_at_vertices[i] - u_at_vertices[i],
                     u_e_at_vertices[i], u_at_vertices[i]))
```

This simple test problem is turned into a real unit test for different function spaces in the function `test_solver_bc`.

Debugging the setting of boundary conditions. It is easy to make mistakes when implementing a problem with many different types of boundary conditions, as in the present case. Some helpful debugging output is to run through all vertex coordinates and check if the `SubDomain.inside` method marks the vertex as on the boundary. Another useful printout is to list which degrees of freedom that are subject to Dirichlet conditions, and for first-order Lagrange elements, add the corresponding vertex coordinate to the output.

```
if debug:
    # Print the vertices that are on the boundaries
```

```
    coor = mesh.coordinates()
    for x in coor:
        if bx0.inside(x, True): print('%s is on x=0' % x)
        if bx1.inside(x, True): print('%s is on x=1' % x)
        if by0.inside(x, True): print('%s is on y=0' % x)
        if by1.inside(x, True): print('%s is on y=1' % x)
    # Print the Dirichlet conditions
    print('No of Dirichlet conditions:', len(bcs))
    d2v = dof_to_vertex_map(V)
    for bc in 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()))))
```

In addition, it is helpful to print the exact and the numerical solution at all the vertices as shown in Section 1.2.4.

Implementation of multiple subdomains. Section 1.4.3 explains how to deal with multiple subdomains of Ω and a piecewise constant coefficient function p that takes on different constant values in the different subdomains. We can easily add this type of p coefficient to the `solver_bc` function. The signature of the function is

```
def solver_bc(
    p, f,                   # Coefficients in the PDE
    boundary_conditions,    # Dict of boundary conditions
    Nx, Ny,                 # Cell division of the domain
    degree=1,               # Polynomial degree
    subdomains=[],          # List of SubDomain objects in domain
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    debug=False,
    ):
...
    return u, p   # p may be modified
```

If `subdomain` is an empty list, we assume there are no subdomains, and p is an `Expression` or `Constant` object specifying a formula for p. If not, `subdomain` is a list of `SubDomain` objects, defining different parts of the domain. The first element is a dummy object, defining "the rest" of the domain. The next elements define specific geometries in the `inside` methods. We start by marking all elements with subdomain number 0, this will then be "the rest" after marking subdomains 1, 2, and so on. The next step is to define `p` as a piecewise constant function over cells and fill it with values. We assume that the user-argument `p` is an array (or list) holding the values of p in the

different parts corresponding to `subdomains`. The returned `p` is needed for flux computations. If there are no subdomains, the returned `p` is just the original `p` argument.

The appropriate code for computing `p` becomes

```
import numpy as np
if subdomains:
    # subdomains is list of SubDomain objects,
    # p is array of corresponding constant values of p
    # in each subdomain
    materials = CellFunction('size_t', mesh)
    materials.set_all(0)  # "the rest"
    for m, subdomain in enumerate(subdomains[1:], 1):
        subdomain.mark(materials, m)

    p_values = p
    V0 = FunctionSpace(mesh, 'DG', 0)
    p  = Function(V0)
    help = np.asarray(materials.array(), dtype=np.int32)
    p.vector()[:] = np.choose(help, p_values)
```

We define $p(x,y) = p_0$ in Ω_0 and $k(x,y) = p_1$ in Ω_1, where $p_0 > 0$ and $p_1 > 0$ are given constants. As boundary conditions, we choose $u = 0$ at $y = 0$, $u = 1$ at $y = 1$, and $\partial u/\partial n = 0$ at $x = 0$ and $x = 1$. One can show that the exact solution is now given by

$$u(x,y) = \begin{cases} \frac{2yp_1}{p_0+p_1}, & y \leq 1/2 \\ \frac{(2y-1)p_0+p_1}{p_0+p_1}, & y \geq 1/2 \end{cases} \tag{1.27}$$

As long as the element boundaries coincide with the internal boundary $y = 1/2$, this piecewise linear solution should be exactly recovered by Lagrange elements of any degree. We can use this property to verify the implementation and make a unit test for a series of function spaces:

```
def test_solvers_bc_2mat():
    tol = 2E-13  # Tolerance for comparisons

    class Omega0(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] <= 0.5+tol

    class Omega1(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] >= 0.5-tol

    subdomains = [Omega0(), Omega1()]
    p_values = [2.0, 13.0]
    boundary_conditions = {
        0: {'Neumann': 0},
        1: {'Neumann': 0},
        2: {'Dirichlet': Constant(0)}, # y=0
        3: {'Dirichlet': Constant(1)}, # y=1
```

```
        }

    f = Constant(0)
    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])

    for Nx, Ny in [(2,2), (2,4), (8,4)]:
        for degree in 1, 2, 3:
            u, p = solver_bc(
                p_values, f, boundary_conditions, Nx, Ny, degree,
                linear_solver='direct', subdomains=subdomains,
                debug=False)

            # Compute max error in infinity norm
            u_e = interpolate(u_exact, u.function_space())
            import numpy as np
            max_error = np.abs(u_e.vector().array() -
                               u.vector().array()).max()
            assert max_error < tol, 'max error: %g' % max_error
```

1.4.5 Refactoring of a solver function into solver and problem 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. The implementation in the previous section required 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, as our mathematical formulation of the problem in the previous section in fact supports, this is not possible without extending the `solver_bc` function.

A different software 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 FEniCS 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_bc` function in the previous section.

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.mesh, degree = problem.mesh_degree()
        self.V = V = FunctionSpace(self.mesh, 'P', degree)
        Dirichlet_cond = 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 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()))))

        u = TrialFunction(V)
        v = TestFunction(V)
        p = problem.p_coeff()
        self.p = p  # store for flux computations
        f = problem.f_rhs()
        F = dot(p*grad(u), grad(v))*dx
        F -= f*v*dx
        F -= sum([g*v*ds_
                  for g, ds_ in problem.Neumann_conditions()])
        F += sum([r*(u-s)*ds_
```

```
                  for r, s, ds_ in problem.Robin_conditions()])
        self.a, self.L = lhs(F), rhs(F)

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

    def solve(self, linear_solver='direct'):
        # Compute solution
        self.u = Function(self.V)

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

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

    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)
        self.flux_u = project(-self.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`.

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()

def test_PoissonSolver():
    """Recover numerial solution to "machine precision"."""
    class TestProblemExact(PoissonProblem):
        def __init__(self, Nx, Ny):
            """Initialize mesh, boundary parts, and p."""
            self.mesh = UnitSquareMesh(Nx, Ny)
            self.u0 = 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.u0

    problem = TestProblemExact(Nx=2, Ny=2)
    problem.solve(linear_solver='direct')
    u = problem.solution()
    u_e = interpolate(problem.u0, 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 `ft08_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 1.4.4? 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.

Chapter 2
The diffusion solver revisited

This chapter is devoted to solving diffusion problems of the form

$$-\nabla \cdot (p\nabla u) = f,$$

with initial condition $u = I$ and various types of Dirichlet, Neumann, and Robin conditions. A very simple FEniCS program for a diffusion equation was introduced in Section **??**, but here we shall discuss important algorithmic optimization strategies, how to store and animate time-dependent data, and how to construct more advanced solvers in terms of classes.

2.1 Optimization of algorithms and implementations

2.1.1 Avoiding some assembly

The time-dependent diffusion equation gives rise to a linear system $AU = b$ at each time level, where the coefficient matrix A is constant, but b depends on u at the previous time level. To increase the computational efficiency, we can therefore assemble A once and for all before the time loop. To be able to do this, we need to explicitly create matrices and vectors as demonstrated in Section 1.2.8.

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

- define Dirichlet boundary condition (u_0, Dirichlet boundary, etc.)
- let u_1 interpolate I or be the projection of I
- define a and L
- assemble matrix A from a

- set some stopping time T
- $t = \Delta t$
- while $t \leq T$
  - assemble vector b from L
  - apply essential boundary conditions
  - solve $AU = b$ for U and store in u
  - $t \leftarrow t + \Delta t$
  - $u_1 \leftarrow u$ (be ready for next step)

The code features the following changes from the `ft02_diffusion_flat1.py` program. We may define a and L from F as before, or do it explicitly:

```
a = u*v*dx + dt*dot(grad(u), grad(v))*dx
L = (u_1 + dt*f)*v*dx
```

Prior to the time loop we assemble the coefficient matrix A once and for all:

```
A = assemble(a)   # assemble only once, before the time stepping
```

At each time level we can do a similar `b = assemble(L)`. 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 set `b = None` such that `b` is defined in the first call to `assemble`.

The necessary changes inside the time loop go as follows:

```
while t <= T:
    b = assemble(L, tensor=b)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)
```

The update `u0.t = t` is of key importance as `bc.apply(A, b)` will look up the `u0` object to find the proper values in the Dirichlet condition, and these change with time in our test problem!

The complete program is found in the file `ft09_diffusion_flat2.py`.

2.1.2 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 $L(v)$ form in (**??**), which in general varies with time through u^{k-1}, f^k, 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, as explained in Section 1.2.8, to identify matrix-vector products that build up the complete system. We have $u^{k-1} = \sum_{j=1}^N U_j^{k-1}\phi_j$, and we can expand f^k as $f^k = \sum_{j=1}^N F_j^k \phi_j$. Inserting these expressions in $L(v)$ and using $v = \hat{\phi}_i$ result in

$$\begin{aligned}\int_\Omega \left(u^{k-1} + \Delta t f^k\right) v \,\mathrm{d}x &= \int_\Omega \left(\sum_{j=1}^N U_j^{k-1}\phi_j + \Delta t \sum_{j=1}^N F_j^k \phi_j\right)\hat{\phi}_i \,\mathrm{d}x, \\ &= \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) U_j^{k-1} + \Delta t \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) F_j^k\,.\end{aligned}$$

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

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

which is nothing but two matrix-vector products,

$$MU^{k-1} + \Delta t M F^k,$$

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

$$U^{k-1} = (U_1^{k-1}, \ldots, U_N^{k-1})^T,$$

and

$$F^k = (F_1^k, \ldots, F_N^k)^T\,.$$

We have immediate access to U^{k-1} in the program since that is the vector in the `u_1` function. The F^k vector can easily be computed by interpolating the prescribed f function (at each time level if f varies with time). Given M, U^{k-1}, and F^k, the right-hand side b can be calculated as

$$b = MU^{k-1} + \Delta t MF^k\,.$$

That is, no assembly is necessary to compute b.

The coefficient matrix A can also be split into two terms. We insert $v = \hat{\phi}_i$ and $u^k = \sum_{j=1}^N U_j^k \phi_j$ in the expression (**??**) to get

$$\sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i \phi_j \,\mathrm{d}x \right) U_j^k + \Delta t \sum_{j=1}^N \left(\int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x \right) U_j^k,$$

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

$$MU^k + \Delta t KU^k = (M + \Delta t K)U^k,$$

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

$$K_{ij} = \int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x\,. \tag{2.1}$$

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

$$a_K(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{2.2}$$

$$a_M(u,v) = \int_\Omega uv \,\mathrm{d}x\,. \tag{2.3}$$

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

Implementation. The following modifications are needed in the `ft10_diffusion_func.py` program from the previous section in order to implement the new strategy of avoiding assembly at each time level:

1. Define separate forms a_M and a_K
2. Assemble a_M to M and a_K to K
3. Compute $A = M + \Delta t K$
4. Define f as an `Expression`
5. Interpolate the formula for f to a finite element function F^k
6. Compute $b = MU^{k-1} + \Delta t MF^k$

The relevant code segments become

```
# 1.
a_K = dot(grad(u), grad(v))*dx
a_M = u*v*dx
# No need for L

# 2. and 3.
M = assemble(a_M)
K = assemble(a_K)
A = M + dt*K

# 4.
f = Expression('beta - 2 - 2*alpha', beta=beta, alpha=alpha)

# 5. and 6.
while t <= T:
    f_k = interpolate(f, V)
    F_k = f_k.vector()
    b = M*u_1.vector() + dt*M*F_k
```

We implement these modifications in a refactored version of the program `ft09_diffusion_flat2.py`, where the solver is a function as explained in Section 1.1 rather than a flat program. The domain can also more flexibly be a 1D, 2D, or 3D interval, rectangle, or box. The new `solver_minimize_assembly` function resides in `ft10_diffusion_func.py`.

```
def solver_minimize_assembly(
    alpha, f, u0, I, dt, T, divisions, L, degree=1,
    user_action=None, I_project=False):
    """
    Solve diffusion PDE u_t = div(alpha*grad(u)) + f on
    an interval, rectangle, or box with side lengths in L.
    divisions reflect cell partitioning, degree the element
    degree. user_action(t, u, timetesp) is a callback function
    where the calling code can process the solution.
    If I_project is false, use interpolation for the initial
    condition.
    """
    # Create mesh and define function space
    d = len(L)  # No of space dimensions
    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)

    class Boundary(SubDomain):  # define the Dirichlet boundary
        def inside(self, x, on_boundary):
            return on_boundary

    boundary = Boundary()
```

```
    bc = DirichletBC(V, u0, boundary)

    # Initial condition
    u_1 = project(I, V) if I_project else interpolate(I, V)
    if user_action is not None:
        user_action(0, u_1, 0)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a_M = u*v*dx
    a_K = alpha*dot(grad(u), grad(v))*dx

    M = assemble(a_M)
    K = assemble(a_K)
    A = M + dt*K
    # Compute solution
    u = Function(V)   # the unknown at a new time level

    b_assemble = 0  # CPU time for assembling all the b vectors
    timestep = 1
    t = dt
    while t <= T:
        t0 = time.clock()
        f_k = interpolate(f, V)
        F_k = f_k.vector()
        b = M*u_1.vector() + dt*M*F_k
        b_assemble += time.clock() - t0
        try:
            u0.t = t
            f.t = t
        except AttributeError:
            pass  # ok if no t attribute in u0
        bc.apply(A, b)
        solve(A, u.vector(), b)

        if user_action is not None:
            user_action(t, u, timestep)
        t += dt
        timestep += 1
        u_1.assign(u)
    #info('total time for assembly of right-hand side: %.2f' % b_assemble)

def application_animate(model_problem):
    import numpy as np, time

    if model_problem == 1:
        # Test problem with exact solution at the nodes also for P1 elements
        alpha = 3; beta = 1.2
        u0 = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                        alpha=alpha, beta=beta, t=0)
        f = Constant(beta - 2 - 2*alpha)
        I = u0
        dt = 0.05; T = 2
```

```
        Nx = Ny = 20
        u_range = [1, 1+1+alpha*1+beta*T]
    elif model_problem == 2:
        # Diffusion of a sin^8 spike
        I = Expression('pow(sin(pi*x[0])*sin(pi*x[1]), 8)')
        f = Constant(0)
        u0 = Constant(0)
        dt = 0.0005; T = 20*dt
        Nx = Ny = 60
        u_range = [0, 1]

    vtkfile = File('diffusion.pvd')

    def animate(t, u, timestep):
        global p
        if t == 0:
            p = plot(u, title='u',
                     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)
        vtkfile << (u, float(t))  # store time-dep Function

    solver_minimize_assembly(
        1.0, f, u0, I, dt, T, (Nx, Ny), (1, 1), degree=2,
        user_action=animate, I_project=False)
```

A special feature in this program is the `user_action` callback function: at every time level, the solution is sent to `user_action`, which is some function provided by the user where the solution can be processed, e.g., stored, analyzed, or visualized. In a unit test for the test example without numerical approximation errors, we can write a call to the solver function,

```
def test_solver():
    import numpy as np
    alpha = 3; beta = 1.2
    u0 = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                    alpha=alpha, beta=beta, t=0)
    f = Constant(beta - 2 - 2*alpha)
    dt = 0.3; T = 1.9
    u0.t = 0

    # Define assert_error callback function
    ...

    solver_minimize_assembly(
        f, u0, u0, dt, T, Nx, Ny, degree,
        user_action=assert_error, I_project=False)
```

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(u0, 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 `ft10_diffusion_func.py`.

Note that `p`, which must survive between subsequent calls to the callback function, has to be declared as a global variable. This is necessary when the user action function is a *closure* (function inside function, "remembering" variables in the parent function) and `p` is changed inside the closure. Some programmers find it more convenient to let the user action be class instead, where `p` can be an attribute. Later examples employ the class design.

The function `solver_vs_solver_minimize_assembly` measures the impact of the optimization technique in this section compared to the simpler technique from the previous section where we need to assemble the right-hand side of the linear system at every time level. The impact is not huge, just a speed-up factor of 1-2 for 2D problems and around 2 for 3D problems. Still, this may be a value factor when you run a code a lot.

2.2 A welding example with post processing and animation

The focus so far in this tutorial has been on producing the solution of PDE problems. 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 animat-

ing 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.

2.2.1 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[1] 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.

2.2.2 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 question 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 $\Omega = [0, L] \times [0, L]$.

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

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.

2.2.3 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. 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:

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

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

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

The first coefficient is a dimensionless number,

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

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 do 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 [2] gives a comprehensive treatment of scaling.

2.2.4 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
    u0 = 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, u0, 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 2.2.1, 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 diffusion_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 diffusion_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`

hpl 11: I AM HERE!

2.3 Refactored implementation

The flat program for the diffusion solver in `ft02_diffusion_flat1.py` and `ft09_diffusion_flat2.py` was refactored in `ft10_diffusion_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.

2.3.1 Mathematical problem

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

$$\varrho c\frac{\partial u}{\partial t} = \nabla\cdot(\kappa\nabla u) + f(\boldsymbol{x},t) \text{ in } \Omega\times(0,T], \tag{2.4}$$

$$u(\boldsymbol{x},0) = I \text{ on } \Omega, \tag{2.5}$$

$$u = u_0(t) \text{ on } \Gamma_D, \tag{2.6}$$

$$-\kappa\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N, \tag{2.7}$$

$$-\kappa\frac{\partial u}{\partial n} = r(u-U_s) \text{ on } \Gamma_R\,. \tag{2.8}$$

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

$$\varrho c\frac{u^{n+1}-u^n}{\Delta t} = \theta(\nabla\cdot\left(\kappa\nabla u^{n+1}\right) + f(\boldsymbol{x},t_{n+1})) + (1-\theta)(\nabla\cdot(\kappa\nabla u^n) + f(\boldsymbol{x},t_n))\,. \tag{2.9}$$

A Galerkin method for this initial-boundary value problem consists of multiplying (2.9) by a test function $v \in \hat{V}$, 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

$$\begin{aligned}F(u;v) = \int_\Omega \big(v\varrho c\frac{u^{n+1}-u^n}{\Delta t}\,\mathrm{d}x + \theta\kappa\nabla u^{n+1}\cdot\nabla v - v\theta f(\boldsymbol{x},t_{n+1})\\ - (1-\theta)\kappa\nabla u^n\cdot\nabla v + v(1-\theta)f(\boldsymbol{x},t_n\big)\,\mathrm{d}x\\ + \int_{\Gamma_N}\big(\theta g(\boldsymbol{x},t_{n+1})v + (1-\theta)g(\boldsymbol{x},t_n)v\big)\,\mathrm{d}s\\ + \int_{\Gamma_R}\big(\theta r(u^{n+1}-U_s(t_{n+1}))v + (1-\theta)r(u^n-U_s(t_n))v\big)\,\mathrm{d}s = 0\,.\end{aligned} \tag{2.10}$$

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,

$$F(u;v) = \int_\Omega \big(v\varrho c\frac{u-u_1}{\Delta t}\,\mathrm{d}x + \theta\kappa\nabla u\cdot\nabla v - v\theta f(\boldsymbol{x},t_{n+1}) \\ -(1-\theta)\kappa\nabla u_1\cdot\nabla v + v(1-\theta)f(\boldsymbol{x},t_n)\big)\,\mathrm{d}x \\ + \int_{\Gamma_N}\big(\theta g(\boldsymbol{x},t_{n+1})v + (1-\theta)g(\boldsymbol{x},t_n)v\big)\,\mathrm{d}s \\ + \int_{\Gamma_R}\big(\theta r(u-U_s(t_{n+1}))v + (1-\theta)r(u_1-U_s(t_n))v\big)\,\mathrm{d}s = 0\,. \quad (2.11)$$

The variational formulation is then: at each time level, find $u\in V$ such that $F(u;v)=0\ \forall v\in\hat{V}$. 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.

2.3.2 A class-based solver

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.

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.

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)

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

        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 a 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')
```

2.3.3 Example: 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).

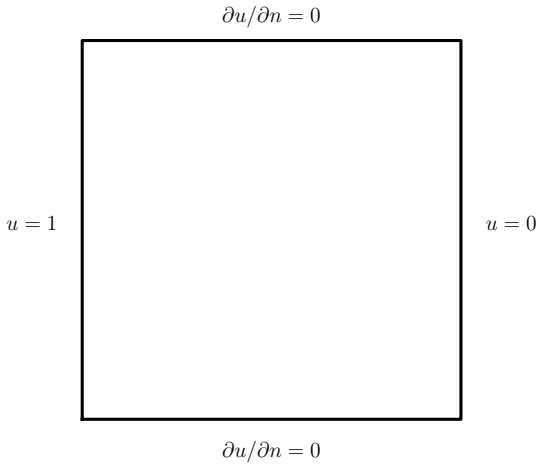


Fig. 2.1 Domain with (scaled) boundary conditions: sudden jump in u at $x = 0$.

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 ($\partial u/\partial n = 0$). The heating is applied to $x = 0$ and $x = L$.

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

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

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

$$\frac{\partial \bar{u}}{\partial \bar{t}} = \bar{\nabla}^2 \bar{u} \text{ in } \Omega = (0,1) \times (0,1) \times (0,T], \tag{2.12}$$

$$\bar{u}(\boldsymbol{x}, 0) = 0 \text{ in } \Omega, \tag{2.13}$$

$$\bar{u} = 1 \text{ at } \Gamma_{D_1}, \tag{2.14}$$

$$\bar{u} = 0 \text{ at } \Gamma_{D_2}, \tag{2.15}$$

$$\frac{\partial \bar{u}}{\partial \bar{n}} = 0 \text{ at } \Gamma_N. \tag{2.16}$$

Implementation. We can solve our problem with the general problem and solver classes by noticing that we set $\varrho = c = \kappa = 1$, 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:

```
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 will gives an animation of the temperature profile, but results in somewhat lengthy code (we actually use SciTools to cut down the length compared to animation code in Matplotlib).

To mark the boundaries, we can make a function like

```
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 abs(x[0] - x0) < tol

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

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

    class BoundaryY1(SubDomain):
        def inside(self, x, on_boundary):
            return on_boundary and abs(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 repeated for 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
```

Now we are in a position to show our problem class:

```
class Problem1(DiffusionProblem):
    """Evolving boundary layer, I=0, but u=1 at x=0."""
    def __init__(self, Nx, Ny):
        DiffusionProblem.__init__(self)
        self.init_mesh(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]
        # scitools animation is simpler than FuncAnimation
        # in matplotlib for the user_action callback function
        import scitools.std as plt
        self.plt = plt
```

```
    def init_mesh(self, Nx, Ny):
        """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)

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

```
                self.plt.plot(
                    x_coor, u_line, '-',
                    legend=['u, t=%.4f' % t],
                    xlabel='x', ylabel='u',
                    axis=[0, 1, 0, 1])
                self.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 demo1():
    problem = Problem1(Nx=20, Ny=5)
    problem.solve(theta=1, linear_solver='direct')
```

Results. Figure 2.2 shows accumulated curves (from `self.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 $\theta = 0$, we never experience any oscillations, but the boundary layer gets thicker and less accurate (smaller Δt is needed to compensate).

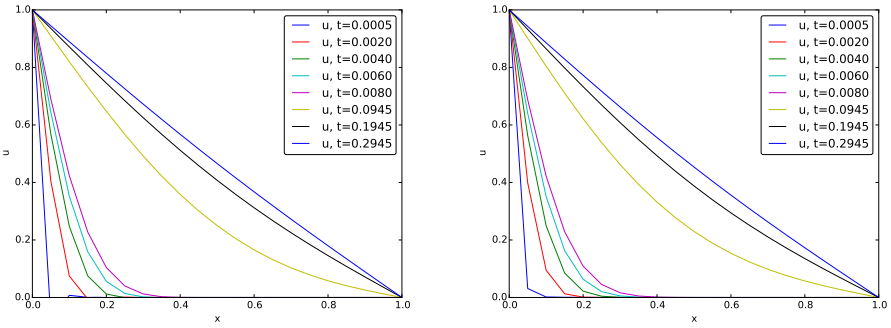


Fig. 2.2 Development of thermal boundary layer: Crank-Nicolson (left) and Backward Euler (right) schemes.

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
```

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`

2.3.4 Extension to 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.

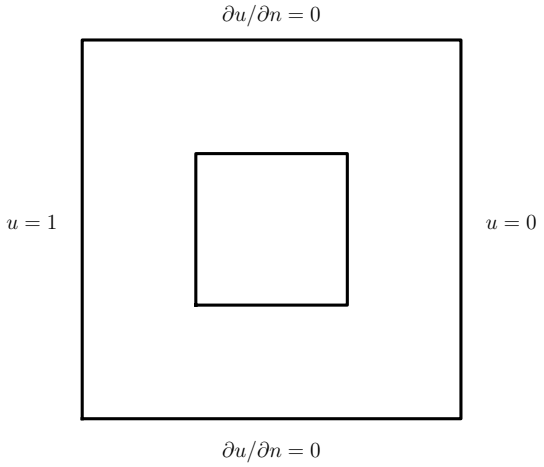


Fig. 2.3 Domain with internal subdomain and (scaled) boundary conditions.

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 12: Would be clearer at more physical code 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):
        Problem1.__init__(self, Nx, Ny)
        self.init_mesh(Nx, Ny, kappa_values)
        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]

    def init_mesh(self, Nx, Ny, kappa_values=[1, 0.1]):
        """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 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.kappa = Function(self.V0)
        help = np.asarray(self.materials.array(), dtype=np.int32)
        self.kappa.vector()[:] = np.choose(help, kappa_values)

    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 demo2():
    problem = Problem2(Nx=20, Ny=5, kappa_values=[1,1000])
    print('kappa:', problem.kappa.vector().array())
    problem.solve(theta=0.5, linear_solver='direct')
```

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.

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`

2.3.5 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:

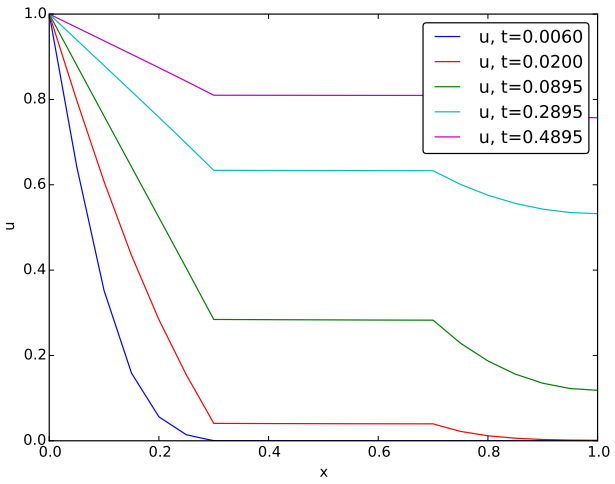


Fig. 2.4 Development of thermal boundary layer in heterogeneous medium.

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

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 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.

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 [2] (see ebook[2]) 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

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

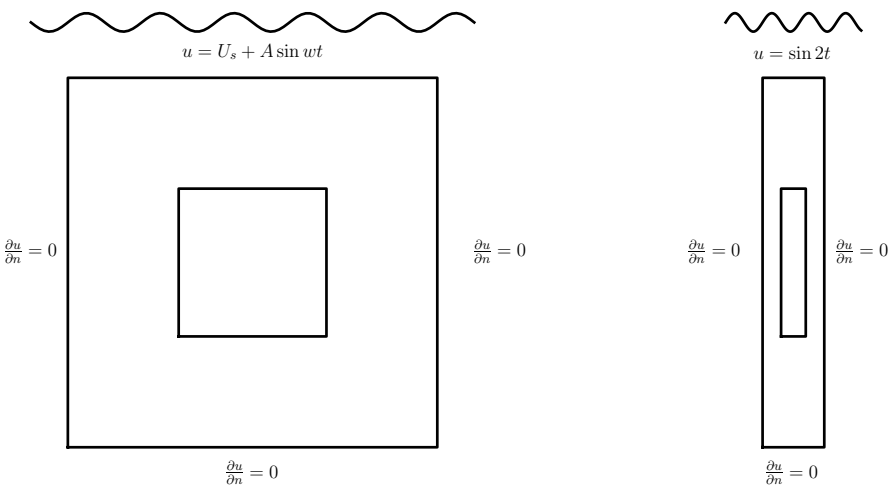


Fig. 2.5 Domain with oscillating temperature at the boundary: unscaled (left) and scaled (left).

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, a typical length of the domain in y direction is 6. The most appropriate 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

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

hpl 13: The position of the subdomain should be scaled. Now it is basically postioned in the scaled domain.

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 $[0,1]\times[0,6]$ as desired. We also need to change the Dirichlet condition. 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):
        # 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)
        # Stretch the mesh in y direction so we get [0,1]x[0,6]
        self.mesh.coordinates()[:] = np.array(
            [self.mesh.coordinates()[:,0],
             6*self.mesh.coordinates()[:,1]]).transpose()

        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=1/2
        y_coor = np.linspace(tol, 6-tol, 101)
        y = [(0.5, y_) for y_ in y_coor]
        u = self.solution()
        u_line = [u(y_) for y_ in y]
        # Animation in figure(1)
        self.plt.figure(1)
        self.plt.plot(
            y_coor, u_line, 'b-',
            legend=['u, t=%.4f' % t],
            title='Solution along x=1/2, time step: %g' %
            self.time_step(t),
            xlabel='y', ylabel='u',
            axis=[0, 6, -1.2, 1.2])
        self.plt.savefig('tmp_%04d.png' % timestep)
        # Accumulated selected curves in one plot in figure(2)
        self.plt.figure(2)
        for t_ in self.times4curveplots:
            if abs(t - t_) < 0.5*self.time_step(t):
                self.plt.plot(
                    y_coor, u_line, '-',
                    legend=['u, t=%.4f' % t],
                    xlabel='y', ylabel='u',
                    axis=[0, 6, -1.2, 1.2])
                self.plt.hold('on')
```

The problem is solved by

```
def demo3():
    problem = Problem3(Nx=5, Ny=20, kappa_values=[1,1000])
    problem.solve(theta=0.5, linear_solver='direct')
```

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.

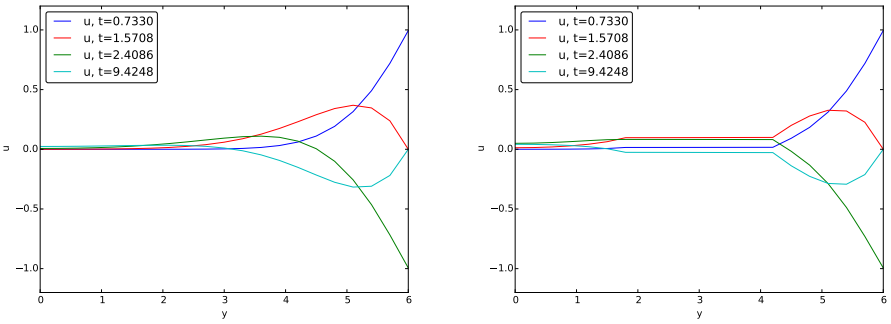


Fig. 2.6 Oscillating boundary temperature: homogeneous (left) and heterogeneous (right) medium.

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`

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
The nonlinear Poisson solver revisited

FEniCS can be used in alternative ways for solving a nonlinear PDE problem. We shall in the following subsections go through three solution strategies:

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

The examples will show how tailored solution algorithms can easily be implemented in a FEniCS solver. (The exposition will hopefully also help newcomers to the finite element method to understand the method better.) However, first we provide more information for using the built-in automated Newton solver that was demonstrated in Section **??**.

Test problem. We continue to address the nonlinear Poisson problem $-\nabla\cdot(q(u)\nabla u) = f$. To be able to easily verify our implementation, we choose the domain, $q(u)$, f, and the boundary conditions such that we have a simple, exact solution u. Let Ω be the unit hypercube $[0,1]^d$ in d dimensions, $q(u) = (1+u)^m$, $f = 0$, $u = 0$ 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 $x_0,\ldots,x_{d-1}$. The exact solution is then

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

We refer to Section 1.2.5 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 **??**. However, in real problems we will at least have control of the tol-

erance 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 (**??**) is straightforwardly defined as follows, assuming `q(u)` is coded as a Python function (see Section **??** 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 UFL 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

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u_- + \epsilon\delta u;v)\,. \tag{3.2}$$

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

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

and obtain

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

which leads to

$$\int_\Omega \nabla v\cdot\left[q'(u_-)\delta u\nabla(u_-)+q(u_-)\nabla\delta u\right]\,\mathrm{d}x, \tag{3.3}$$

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 having the advantage that we easily recognize the expression 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
```

The alternative specification of `F`, with `u` as `TrialFunction`, leads to

```
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 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 `ft12_np_anyD.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,
    TrialFunction_object='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

Picard iteration 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*,

$$\nabla \cdot \left(q(u^k) \nabla u^{k+1} \right) = 0, \quad k = 0, 1, \ldots \tag{3.4}$$

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.

We can easily formulate a variational problem for u^{k+1} from (3.4). Equivalently, we can approximate $q(u)$ by $q(u^k)$ in (**??**) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u^{k+1} \in V$ such that

$$\tilde{F}(u^{k+1}; v) = 0 \quad \forall v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{3.5}$$

with

$$\tilde{F}(u^{k+1}; v) = \int_\Omega q(u^k) \nabla u^{k+1} \cdot \nabla v \, \mathrm{d}x \,. \tag{3.6}$$

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

$$a(u^{k+1}, v) = L(v), \tag{3.7}$$

with

$$a(u, v) = \int_\Omega q(u^k) \nabla u \cdot \nabla v \, \mathrm{d}x \tag{3.8}$$

$$L(v) = 0 \,. \tag{3.9}$$

The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| < \text{tol}$, 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.

In the solution algorithm we only need to store u^k and u^{k+1}, called `u_k` and `u` in the code below. 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_k = interpolate(Constant(0.0), V)  # previous (known) u
a = dot(q(u_k)*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_k||
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)
    diff = u.vector().array() - u_k.vector().array()
    eps = numpy.linalg.norm(diff, ord=numpy.Inf)
    print('iter=%d: norm=%g' % (iter, eps))
    u_k.assign(u)   # update for next iteration
```

We need to define the previous solution in the iterations, `u_k`, as a finite element function so that `u_k` can be updated with `u` at the end of the loop. We may create the initial `Function u_k` 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 `ft13_picard_np.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 1.2.5. 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

After having discretized our nonlinear PDE problem, we may use Newton's method to solve the system of nonlinear algebraic equations. From the continuous variational problem (**??**), the discrete version (**??**) results in 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 (**??**)):

$$F_i(U_1,\ldots,U_N) \equiv \sum_{j=1}^{N} \int_\Omega \left(q\left(\sum_{\ell=1}^{N} U_\ell \phi_\ell\right) \nabla\phi_j U_j \right) \cdot \nabla\hat{\phi}_i \,\mathrm{d}x = 0, \quad i = 1,\ldots,N\,. \tag{3.10}$$

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

$$\sum_{j=1}^{N} \frac{\partial}{\partial U_j} F_i(U_1^k,\ldots,U_N^k)\delta U_j = -F_i(U_1^k,\ldots,U_N^k), \quad i = 1,\ldots,N, \tag{3.11}$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \quad j = 1,\ldots,N, \tag{3.12}$$

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 (3.10). The derivative $\partial F_i/\partial U_j$ becomes

$$\int_\Omega \left[q'(\sum_{\ell=1}^{N} U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^{N} U_j^k \phi_j) \cdot \nabla\hat{\phi}_i + q\left(\sum_{\ell=1}^{N} U_\ell^k \phi_\ell\right) \nabla\phi_j \cdot \nabla\hat{\phi}_i \right] \mathrm{d}x\,. \tag{3.13}$$

The following results were used to obtain (3.13):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j}\sum_{j=1}^{N} U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j} q(u) = q'(u)\phi_j\,. \tag{3.14}$$

We can reformulate the Jacobian matrix in (3.13) by introducing the short notation $u^k = \sum_{j=1}^{N} U_j^k \phi_j$:

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

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 to be solved in each Newton iteration:

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

This variational form fits the standard notation $a(\delta u, v) = L(v)$ with

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

Note the important feature in Newton's method that the previous solution u^k 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.

We now turn to the 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 recipe for solving this problem appears in Sections **??**, **??**, and 1.4.1. The code for computing 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)
a = dot(grad(u), grad(v))*dx
f = Constant(0.0)
L = f*v*dx
A, b = assemble_system(a, L, bcs)
u_k = Function(V)
U_k = u_k.vector()
solve(A, U_k, b)
```

Here, `u_k` denotes the solution function for the previous iteration, so that the solution after each Newton iteration is `u = u_k + omega*du`. Initially, `u_k` is the initial guess we call 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)

du = TrialFunction(V) # u = u_k + omega*du
a = dot(q(u_k)*grad(du), grad(v))*dx + \
    dot(Dq(u_k)*du*grad(u_k), grad(v))*dx
L = -dot(q(u_k)*grad(u_k), grad(v))*dx
```

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_k + omega*du
omega = 1.0       # relaxation parameter
eps = 1.0
tol = 1.0E-5
iter = 0
maxiter = 25
while eps > tol and iter < maxiter:
    iter += 1
    A, b = assemble_system(a, L, bcs_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print('Norm:', eps)
    u.vector()[:] = u_k.vector() + omega*du.vector()
    u_k.assign(u)
```

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

```
u.assign(u_k)  # u = u_k
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.

Mesh construction for a d-dimensional problem with arbitrary degree of the Lagrange elements can be done as explained in Section 1.2.5. The complete program appears in the file `ft14_alg_newton_np.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.

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,

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

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:

$$\int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{3.20}$$

The variational problem reads: find $\delta u \in V$ such that $a(\delta u, v) = L(v)$ for all $v \in \hat{V}$, where

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

$$L(v) = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \,\mathrm{d}x\,. \tag{3.22}$$

The function spaces V and $\hat{V}$, being continuous or discrete, are as in the linear Poisson problem from Section **??**.

We must provide some initial guess, e.g., the solution of the PDE with $q(u) = 1$. 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 $a(\delta u, v) = L(v)$ for δu and compute a new approximation $u^{k+1} = u^k + \delta u$. Note that δu is a correction, so if u^0 satisfies the prescribed Dirichlet conditions on some part Γ_D of the boundary, we must demand $\delta u = 0$ on Γ_D.

Looking at (3.21) and (3.22), we see 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 `ft15_pde_newton_np.py`. The reader is encouraged to go through this code to be convinced that the present method actually ends up with the same program as needed for the Newton method at the linear algebra level in Section 3.2.2.

Chapter 4
More old stuff

4.1 Creating more complex domains

Up to now we have been very fond of the unit square as domain, which is an appropriate choice for initial versions of a PDE solver. The strength of the finite element method, however, is its ease of handling domains with complex shapes. This section shows some methods that can be used to create different types of domains and meshes.

Domains of complex shape must normally be constructed in separate preprocessor programs. Two relevant preprocessors are Triangle for 2D domains and NETGEN for 3D domains.

4.1.1 Built-in mesh generation tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitIntervalMesh`, `UnitSquareMesh`, `UnitCubeMesh`, `IntervalMesh`, `RectangleMesh`, and `BoxMesh`. Some of these names have been briefly met in previous sections. The hopefully self-explanatory code snippet below summarizes typical constructions of meshes with the aid of these tools:

```
# 1D domains
mesh = UnitIntervalMesh(20)     # 20 cells, 21 vertices
mesh = IntervalMesh(20, -1, 1)  # domain [-1,1]

# 2D domains (6x10 divisions, 120 cells, 77 vertices)
mesh = UnitSquareMesh(6, 10)  # 'right' diagonal is default
# The diagonals can be right, left or crossed
mesh = UnitSquareMesh(6, 10, 'left')
mesh = UnitSquareMesh(6, 10, 'crossed')
```

```
# Domain [0,3]x[0,2] with 6x10 divisions and left diagonals
mesh = RectangleMesh(0, 0, 3, 2, 6, 10, 'left')

# 6x10x5 boxes in the unit cube, each box gets 6 tetrahedra:
mesh = UnitCubeMesh(6, 10, 5)

# Domain [-1,1]x[-1,0]x[-1,2] with 6x10x5 divisions
mesh = BoxMesh(-1, -1, -1, 1, 0, 2, 6, 10, 5)
```

4.1.2 Transforming mesh coordinates

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,

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^s \tag{4.1}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{4.2}$$

toward $x = b$. Figure 4.1 shows the effect of making a rectangular mesh denser toward $x = 0$ (prior to the coordinate transformation below).

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 4.1.

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.

4.2 A General d-Dimensional multi-material test problem

This section is in a preliminary state!

The purpose of the present section is to generalize the basic ideas from the previous section to a problem involving an arbitrary number of materials in 1D, 2D, or 3D domains. The example also highlights how to build more general and flexible FEniCS applications.

More to be done:

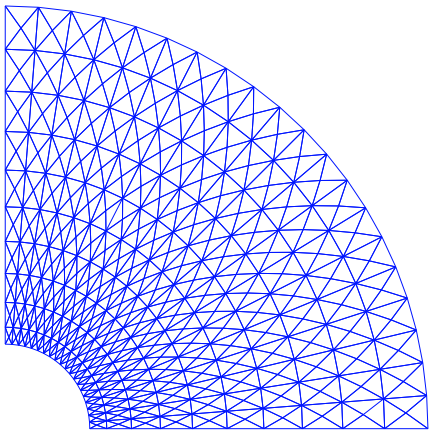


Fig. 4.1 Hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

- Batch compilation of subdomains, see mailinglist.txt, lots of useful stuff in Hake's example with "pointwise", see what the bcs are etc.
- Use of `near` or similar function (better: user-adjusted tolerance)

4.2.1 The PDE problem

We generalize the problem in Section 1.4.3 to the case where there are s materials $\Omega_0, \ldots, \Omega_{s-1}$, with associated constant k values $k_0, k_1, \ldots, k_{s-1}$, as illustrated in Figure 4.2.

Although the sketch of the domain is in two dimensions, we can easily define this problem in any number of dimensions, using the ideas of Section 1.2.5, but the layer boundaries are planes $x_0 = \text{const}$ and u varies with x_0 only.

The PDE reads

$$\nabla \cdot (k \nabla u) = 0\,. \tag{4.3}$$

To construct a problem where we can find an analytical solution that can be computed to machine precision regardless of the element size, we choose Ω as a hypercube $[0,1]^d$, and the materials as layers in the x_0 direction, as depicted in Figure 4.2 for a 2D case with four materials. The boundaries $x_0 = 0$ and $x_0 = 1$ have Dirichlet conditions $u = 0$ and $u = 1$, respectively,

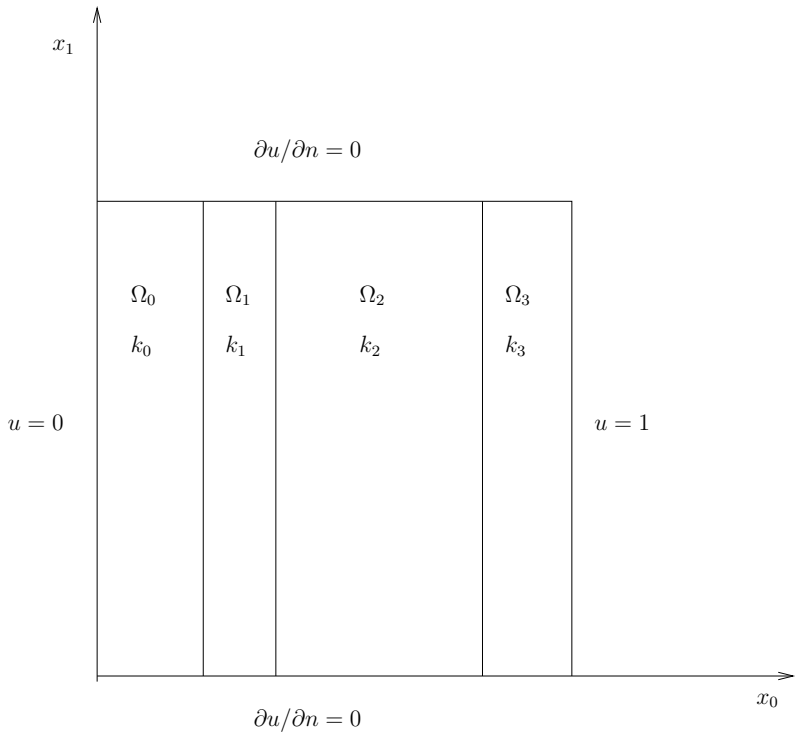


Fig. 4.2 Sketch of a multi-material problem.

while Neumann conditions $\partial u/\partial n = 0$ are set on the remaining boundaries. The complete boundary-value problem is then

$$\begin{aligned} \nabla \cdot (k(x_0)\nabla u(x_0,\ldots,x_{d-1})) &= 0 \text{ in } \Omega, \\ u &= 0 \text{ on } \Gamma_0, \\ u &= 1 \text{ on } \Gamma_1, \\ \tfrac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_N\,. \end{aligned} \tag{4.4}$$

The domain Ω is divided into s materials Ω_i, $i = 0,\ldots,s-1$, where

$$\Omega_i = \{(x_0,\ldots,x_{d-1}) \,|\, L_i \le x_0 < L_{i+1}\}$$

for given x_0 values $0 = L_0 < L_1 < \cdots < L_s = 1$ of the material (subdomain) boundaries. The $k(x_0)$ function takes on the value k_i in Ω_i.

The exact solution of the basic PDE in (4.4)

$$u(x_0,\ldots,x_{d-1}) = \frac{\int_0^{x_0}(k(\tau))^{-1}d\tau}{\int_0^1(k(\tau))^{-1}d\tau}\,.$$

For a piecewise constant $k(x_0)$ as explained, we get

$$u(x_0,\ldots,x_{d-1}) = \frac{(x_0 - L_i)k_i^{-1} + \sum_{j=0}^{i-1}(L_{j+1}-L_j)k_j^{-1}}{\sum_{j=0}^{s-1}(L_{j+1}-L_j)k_j^{-1}}, \quad L_i \leq x_0 \leq L_{i+1}\,. \tag{4.5}$$

That is, $u(x_0,\ldots,x_{d-1})$ is piecewise linear in x_0 and constant in all other directions. If L_i coincides with the element boundaries, Lagrange elements will reproduce this exact solution to machine precision, which is ideal for a test case.

4.2.2 Preparing a mesh with subdomains

Our first task is to generate a mesh for $\Omega = [0,1]^d$ and divide it into subdomains

$$\Omega_i = \{(x_0,\ldots,x_{d-1}) \,|\, L_i < x_0 < L_{i+1}\}$$

for given subdomain boundaries $x_0 = L_i$, $i = 0,\ldots,s$, $L_0 = 0$, $L_s = 1$. Note that the boundaries $x_0 = L_i$ are points in 1D, lines in 2D, and planes in 3D.

Let us, on the command line, specify the polynomial degree of Lagrange elements and the number of element divisions in the various space directions, as explained in detail in Section 1.2.5. This results in an object `mesh` representing the interval $[0,1]$ in 1D, the unit square in 2D, or the unit cube in 3D.

Specification of subdomains (and boundary parts, if desired) is done using a user-defined subclass of `SubDomain`, as explained in Section 1.4.3. We could, in principle, introduce one subclass of `SubDomain` for each subdomain, and this would be feasible if one has a small and fixed number of subdomains as in the example in Section 1.4.3 with two subdomains. Our present case is more general as we have s subdomains. It then makes sense to create one subclass `Material` of `SubDomain` and have an attribute to reflect the subdomain (material) number. We use this number in the test whether a spatial point `x` is inside a subdomain or not:

```
class Material(SubDomain):
    """Define material (subdomain) no. i."""
    def __init__(self, subdomain_number, subdomain_boundaries):
        self.number = subdomain_number
        self.boundaries = subdomain_boundaries
        SubDomain.__init__(self)

    def inside(self, x, on_boundary):
        i = self.number
        L = self.boundaries         # short form (cf. the math)
        if L[i] <= x[0] <= L[i+1]:
            return True
```

```
        else:
            return False
```

The `<=` in the test if a point is inside a subdomain is important as `x` will equal vertex coordinates in the cells, and all vertices of a cell must lead to a `True` return value from the `inside` method for the cell to be a part of the actual subdomain. That is, the more mathematically natural test `L[i] <= x[0] < L[i+1]` fails to include elements with $x = L_{i+1}$ as boundary in subdomain Ω_i.

The marking and numbering of all subdomains goes as follows:

```
cell_entity_dim = mesh.topology().dim()  # = d
subdomains = MeshFunction('uint', mesh, cell_entity_dim)
# Mark subdomains with numbers i=0,1,\ldots,s (=len(L)-1)
for i in range(s):
    material_i = Material(i, L)
    material_i.mark(subdomains, i)
```

We have now all the geometric information about subdomains in a `MeshFunction` object `subdomains`. The subdomain number of mesh entity number `e`, here cell `e`, is given by `subdomains.array()[e]`.

The code presented so far had the purpose of preparing a mesh and a mesh function defining the subdomain. It is smart to put this code in a separate file, say `ft16_define_layers.py`, and view the code as a preprocessing step. We must then store the computed mesh and mesh function in files. Another program may load the files and perform the actually solve the boundary-value problem.

Storing the mesh itself and the mesh function in XML format is done by

```
file = File('hypercube_mesh.xml.gz')
file << mesh
file = File('layers.xml.gz')
file << subdomains
```

This preprocessing code knows about the layer geometries and the corresponding k, which must be propagated to the solver code. One idea is to let the preprocessing code write a Python module containing the `L` and `k` lists as well as an implementation of a function that evaluates the exact solution. The solver code can import this module to get access to `L`, `k`, and the exact solution (for comparison). The relevant Python code for generating a Python module may take the form

```
f = open('u_layered.py', 'w')
f.write("""
import numpy
L = numpy.array(
#s, float)
k = numpy.array(
#s, float)
s = len(L)-1
```

```
def u_e(x):
    # First find which subdomain x0 is located in
    for i in range(len(L)-1):
        if L[i] <= x <= L[i+1]:
            break

    # Vectorized implementation of summation:
    s2 = sum((L[1:s+1] - L[0:s])*(1.0/k[:]))
    if i == 0:
        u = (x - L[i])*(1.0/k[0])/s2
    else:
        s1 = sum((L[1:i+1] - L[0:i])*(1.0/k[0:i]))
        u = ((x - L[i])*(1.0/k[i]) + s1)/s2
    return u

if __name__ == '__main__':
    # Plot the exact solution
    from scitools.std import linspace, plot, array
    x = linspace(0, 1, 101)
    u = array([u_e(xi) for xi in x])
    print(u)
    plot(x, u)
"""
# (L, k))
f.close()
```

4.2.3 Solving the PDE problem

The solver program starts with loading a prepared mesh with a mesh function representing the subdomains:

```
mesh = Mesh('hypercube_mesh.xml.gz')
subdomains = MeshFunction('uint', mesh, 'layers.xml.gz')
```

The next task is to define the k function as a finite element function. As we recall from Section 1.4.3, a k that is constant in each element is suitable. We then follow the recipe from Section 1.4.3 to compute k:

```
V0 = FunctionSpace(mesh, 'DG', 0)
k = Function(V0)

# Vectorized calculation
help = numpy.asarray(subdomains.array(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The essential boundary conditions are defined in the same way is in `dn2_p2D.py` from Section 1.4.2 and therefore not repeated here. The variational problem is defined and solved in a standard manner,

```
u = TrialFunction(V)
```

```
v = TestFunction(V)
f = Constant(0)
a = k*dot(grad(u), grad(v))*dx
L = f*v*dx

problem = VariationalProblem(a, L, bc)
u = problem.solve()
```

Plotting the discontinuous `k` is often desired. Just a `plot(k)` makes a continuous function out of `k`, which is not what we want. Making a `MeshFunction` over cells and filling in the right k values results in an object that can be displayed as a discontinuous field. A relevant code is

```
k_meshfunc = MeshFunction('double', mesh, mesh.topology().dim())

# Scalar version
for i in range(len(subdomains.array())):
    k_meshfunc.array()[i] = k_values[subdomains.array()[i]]

# Vectorized version
help = numpy.asarray(subdomains.array(), dtype=numpy.int32)
k_meshfunc.array()[:] = numpy.choose(help, k_values)

plot(k_meshfunc, title='k as mesh function')
```

The file `ft17_Poisson_layers.py` contains the complete code.

4.3 More Examples

Many more topics could be treated in a FEniCS tutorial, e.g., how to solve systems of PDEs, how to work with mixed finite element methods, how to create more complicated meshes and mark boundaries, and how to create more advanced visualizations. However, to limit the size of this tutorial, the examples end here. There are, fortunately, a rich set of FEniCS demos. The FEniCS documentation explains a collection of PDE solvers in detail: the Poisson equation, the mixed formulation for the Poission equation, the Biharmonic equation, the equations of hyperelasticity, the Cahn-Hilliard equation, and the incompressible Navier-Stokes equations. Both Python and C++ versions of these solvers are explained. An eigenvalue solver is also documented. In the `fenics/demo` directory of the DOLFIN source code tree you can find programs for these and many other examples, including the advection-diffusion equation, the equations of elastodynamics, a reaction-diffusion equation, various finite element methods for the Stokes problem, discontinuous Galerkin methods for the Poisson and advection-diffusion equations, and an eigenvalue problem arising from electromagnetic waveguide problem with Nedelec elements. There are also numerous demos on how to apply various functionality in FEniCS, e.g., mesh refinement and error control, moving meshes (for ALE

methods), computing functionals over subsets of the mesh (such as lift and drag on bodies in flow), and creating separate subdomain meshes from a parent mesh.

The project cbc.solve (`https://launchpad.net/cbc.solve`) offers more complete PDE solvers for the Navier-Stokes equations, the equations of hyperelasticity, fluid-structure interaction, viscous mantle flow, and the bidomain model of electrophysiology. Most of these solvers are described in the "FEniCS book" [3] (`https://launchpad.net/fenics-book`). Another project, cbc.rans (`https://launchpad.net/cbc.rans`), offers an environment for very flexible and easy implementation of Navier-Stokes solvers and turbulence [4, 5]. For example, cbc.rans contains an elliptic relaxation model for turbulent flow involving 18 nonlinear PDEs. FEniCS proved to be an ideal environment for implementing such complicated PDE models. The easy construction of systems of nonlinear PDEs in cbc.rans has been further generalized to simplify the implementation of large systems of nonlinear PDEs in general. The functionality is found in the cbc.pdesys package (`https://launchpad.net/cbcpdesys`).

4.4 Miscellaneous topics

hpl 14: Needs to be cleaned up.

4.4.1 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 1.3.1).

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`

4.4.2 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 4.1.

`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 **??**) 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 1.4.3) or for boundary conditions (see Section 1.4.4).

`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`.

4.4.3 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.

4.4.4 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 5
Troubleshooting

5.1 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.

5.1.1 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

```
instant-clean
```

in a terminal window whenever you encounter a compilation problem.

5.1.2 Syntax errors in expressions

If the compilation problem arises from line with an `Expression` object, examine the syntax of the expression carefully. Section **??** 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`.

5.1.3 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)`.

5.1.4 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.

5.1.5 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`.

5.1.6 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.

5.1.7 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.

5.2 Problems with Expression Objects

5.2.1 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).

5.2.2 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.

5.3 Other Problems

5.3.1 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).

5.3.2 The plot disappears quickly from the screen

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

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.4 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` `ft06_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 map the solution to a structured grid with structured data, i.e., a `BoxField`, see Chapter **??**, and then examine the solution field along grid lines in x and y directions. For example, you can easily check that correct Dirichlet conditions are set, e.g., at the upper boundary (check `u_box[:,-1]`).

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.

5.5 To-do list

5.5.1 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_flat.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.

5.5.2 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.
- 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.
- Meshr domains. Set up some common continuum mechanics examples first. (HPL/AL)
- Decide on an elasticity problem.
- Simple Navier-Stokes solver. Do backward facing step and flow around a cylinder.
- 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 :-)
- Show how the definition of boundaries can be done via strings compiled to C++ (as soon as we have an example with non-trivial boundary segments), cf. Navier-Stokes FEniCS demo.
- 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.
- Should we say "in this book" or "in this tutorial"?
- 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 3.510^{-15},

so `DOLFIN_EPS` is strictly for scaled problems only, where all variables are in $[0, 1]$.

5.5.3 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'
```

Contents

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 G. K. Pedersen. *Scaling of Differential Equations*. Simula Springer Brief Series. Springer, 2016. `http://tinyurl.com/qfjgxmf/web`.

[3] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Partial Differential Equations by the Finite Element Method*. Springer, 2012.

[4] M. Mortensen, H. P. Langtangen, and J. Myre. Cbc.rans - a new flexible, programmable software framework for computational fluid dynamics. In H. I. Andersson and B. Skallerud, editors, *Sixth National Conference on Computational Mechanics (MekIT'11)*. Tapir, 2011.

[5] M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations. *Advances in Water Resources*, 2011. doi: 10.1016/j.advwatres.2011.02.013.

Index