Logg
Mardal
Wells
(Eds.)

Automated Scientific Computing

Automated Scientific Computing
Logg, Mardal, Wells (editors)

This is a preliminary draft, May 7, 2010.
Send comments and suggestions to `fenics-book-dev@fenics.org`.

Automated Scientific Computing

The FEniCS Book

Preliminary draft, May 7, 2010

And now that we may give final praise to the machine we may say that it will be desirable to all who are engaged in computations which, it is well known, are the managers of financial affairs, the administrators of others' estates, merchants, surveyors, geographers, navigators, astronomers...For it is unworthy of excellent men to lose hours like slaves in the labor of calculations which could safely be relegated to anyone else if the machine were used.

Gottfried Wilhelm Leibniz (1646–1716)

Contents

CHAPTER 1

Introduction

By Anders Logg, Garth N. Wells and Kent-Andre Mardal

Chapter ref: **[intro]**

CHAPTER 2

A FEniCS Tutorial

By Hans Petter Langtangen

Chapter ref: **[langtangen]**

2.1 The Fundamentals

FEniCS is a user-friendly tool for solving partial differential equations (PDEs). The purpose of this tutorial is get you started with solving PDEs with the aid of FEniCS. First, we present a series of simple examples and demonstrate

- how to define the PDE problem in terms of a variational problem
- how to define simple domains
- how to deal with Dirichlet, Neumann, and Robin conditions
- how to treat variable coefficients
- how to deal with domains built of several materials (subdomains)
- how to compute derived quantities like the flux vector field or a functional of the solution
- how to quickly visualize the mesh, the solution, the flux, etc.
- how to solve nonlinear PDEs in various ways
- how to deal with time-dependent PDEs

- how to set parameters governing solution methods for linear systems
- how to create domains of more complex shape

The mathematics of the illustrations is kept simple to better focus on FEniCS functionality and syntax. This means that we mostly use the Poisson equation and the time-dependent diffusion equation as model problems, often with input data adjusted such that we get a very simple solution that can be exactly reproduced by any standard finite element method over a uniform, structured mesh. This latter property greatly enhances the verification of the impelementations. Occasionally we insert a physically more relevant example to remind the reader that changing the PDE and boundary conditions to something more real might often be a trivial task.

FEniCS may seem to require a thorough understanding of the abstract mathematical version of the finite element method as well as familiarity with the Python programming language. Nevertheless, it turns out that many are able to pick up the fundamentals of finite elements *and* Python programming as they go along with this tutorial. Simply keep on reading and try out the examples. You will be amazed of how easy it is to solve PDEs with FEniCS!

Reading this tutorial obviously requires access to a machine where the FEniCS software is installed. Chapter 2.8.3 explains briefly how to install the necessary tools.

2.1.1 The Poisson Equation

Computer programming books frequently start with an example on how to print "Hello, World!" on the screen. The counterpart to the "Hello, World!" example in the world of software for partial differential equations is a program which solves the Poisson problem,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_0 \quad \text{on } \partial\Omega\,. \end{aligned} \tag{2.1}$$

Here, $u(\boldsymbol{x})$ is the unknown function, $f(\boldsymbol{x})$ is a prescribed function of space, Δ is the Laplace operator (also often written as ∇^2), Ω is the spatial domain, and $\partial\Omega$ is the boundary of Ω. A stationary PDE like this, together with a complete set of boundary conditions, constitute a *boundary-value problem*, which must be precisely stated before it makes sense to start solving it with FEniCS.

In two space dimensions with coordinates x and y, we can write out the Poisson equation (2.1) in detail:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)\,. \tag{2.2}$$

The unknown u is now a function of two variables, $u(x, y)$, defined over a two-dimensional domain Ω.

The Poisson equation (2.1) arises in numerical physical contexts, for example, heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, water waves. Moreover, the equation appears in numerical splitting strategies of more complicated systems of PDEs, in particular the Navier-Stokes equations.

2.1.2 Variational Formulation

FEniCS makes it easy to solve PDEs if finite elements are used for discretization in space and the problem is expressed as a *variational problem*. Readers who are not familiar with variational problems will get a brief introduction to the topic in this tutorial, and in Chapter **??**, but we encourage getting and reading a proper book on the finite element method in addition. Chapter 2.8.4 contains a list of some suitable books.

The core of the recipe for turning a PDE into a variational problem is to multiply the PDE by a function v, integrate the resulting equation over Ω, and perform integration by parts of terms with second-order derivatives. The function v which multiplies the PDE is in the mathematical finite element literature called a *test function*. The unknown function u to be approximated is referred to as a *trial function*. The terms test and trial function are used in FEniCS programs too. Suitable function spaces must be specified for the test and trial functions. For standard PDEs arising in physics and mechanics such spaces are well known.

In the present case, we first multiply by the test function v and integrate,

$$-\int_\Omega v\Delta u \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x\,. \tag{2.3}$$

Then we apply integration by parts of the integrand with second-order derivatives,

$$-\int_\Omega v\Delta u \,\mathrm{d}x = \int_\Omega \nabla v\cdot\nabla u \,\mathrm{d}x - \int_{\partial\Omega} v\frac{\partial u}{\partial n}\,\mathrm{d}s. \tag{2.4}$$

The test function v is required to vanish on the parts of the boundary where u is known, which in the present problem implies that $v = 0$ on the whole boundary $\partial\Omega$. The second term on the right-hand side of (2.4) therefore vanishes. From (2.3) and (2.4) it follows that

$$\int_\Omega \nabla v\cdot\nabla u \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x\,. \tag{2.5}$$

This equation is supposed to hold for all v in some function space $\hat{V}$. The trial function u lies in some (possible other) function space V. We refer to (2.5) as the *weak form* of the original boundary-value problem (2.1).

The proper statement of our variational problem now goes as follows: Find $u \in V$ such that

$$\int_\Omega \nabla v\cdot\nabla u \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x \quad \forall v\in\hat{V}. \tag{2.6}$$

The test and trial spaces $\hat{V}$ and V are in the present problem defined as

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

In short, $H^1(\Omega)$ is the mathematically well-known Sobolev space containing functions v such that v^2 and $||\nabla v||^2$ have finite integrals over Ω. This implies that the functions must be continuous, but the derivatives can be discontinuous. The solution of the underlying PDE, on the contrary, must lie in a function space where also the derivatives are continuous. The weaker continuity requirements of the variational statement (2.6), caused by the integration by parts, have great practical consequences when it comes to constructing finite elements.

To solve the Poisson equation numerically, we need to transform the continuous variational problem (2.6) to a discrete variational problem. This is done by introducing *finite-dimensional* test and trial spaces $\hat{V}_h \subset \hat{V}$ and $V_h \subset V$. The discrete variational problem reads: Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla v \cdot \nabla u_h \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x \quad \forall v \in \hat{V}_h \subset \hat{V}\,. \tag{2.7}$$

The choice of $\hat{V}_h$ and V_h follows directly from the kind of finite elements we want to apply in our problem. For example, choosing the well-known linear triangular element with three nodes implies that $\hat{V}_h$ and V_h are the spaces of all piecewise linear functions over a mesh of triangles, where the functions in $\hat{V}_h$ are zero on the boundary and those in V_h equal u_0 on the boundary.

The mathematics literature on variational problems applies u_h for the solution of the discrete problem and u for the solution of the continous problem. To obtain (almost) a one-to-one relationshop between the mathematical formulation of a problem and the corresponding FEniCS program, we shall use u for the solution of the discrete problem and u_e for the exact solution of the continuous problem, if we need to explicitly distinguish between the two. In most cases we will introduce the PDE problem with u as unknown and then simply let u denote the finite element solution when we come to the discrete variational problem and the associated program development.

It turns out to be convenient to introduce a unified notation for a weak form like (2.7):

$$a(u, v) = L(v)\,. \tag{2.8}$$

In the present problem we have that

$$a(u, v) = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x, \tag{2.9}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x\,. \tag{2.10}$$

From the mathematics literature, $a(u, v)$ is known as a *bilinear form* and $L(u)$ as a *linear form*. We shall in every problem we solve identify the terms with the unknown u and collect them in $a(u, v)$, and similarly collect all terms with only known functions in $L(v)$. The formulas for a and L are then coded directly in the program.

To summarize, before making a FEniCS program for solving a PDE, we must first perform two steps:

1. Turn the PDE problem into a discrete variational problem: Find $u \in V_h$ such that
$$a(u, v) = L(v) \quad \forall v \in \hat{V}_h \,.$$
2. Specify the choice of discrete spaces, i.e., choice of finite elements.

2.1.3 The Implementation

The test problem so far has a general domain Ω and general functions u_0 and f. However, we must specify Ω, u_0, and f prior to our first implementation. It will be wise to construct a specific problem where we can easily check that the solution is correct. Let us choose $u(x, y) = 1 + x^2 + 2y^2$ to be the solution of our Poisson problem since the finite element method with linear elements over a uniform mesh of triangular cells should exactly reproduce a second-order polynomial at the vertices of the cells, regardless of the size of the elements. This property allows us to verify the code by using very few elements and checking that the computed and the exact solution equal to machine precision. Test problems with this property will be frequently constructed throughout the present tutorial.

Specifying $u(x, y) = 1 + x^2 + 2y^2$ in the problem from Chapter 2.1.2 implies $u_0(x, y) = 1 + x^2 + 2y^2$ and $f(x, y) = -6$. We let Ω be the unit square for simplicity. A FEniCS program for solving (2.1) with the given choices of u_0, f, and Ω may look as follows (the complete code can be found in the file `Poisson2D_D1.py`):

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(6, 4)
V = FunctionSpace(mesh, 'CG', 1)

# Define boundary conditions
u0 = Function(V, '1 + x[0]*x[0] + 2*x[1]*x[1]')

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

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

# Define variational problem
v = TestFunction(V)
```

```
u = TrialFunction(V)
f = Constant(mesh, -6.0)
a = dot(grad(v), grad(u))*dx
L = v*f*dx

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

# Plot solution and mesh
plot(u)
plot(mesh)

# Dump solution to file in VTK format
file = File('poisson.pvd')
file << u

# Hold plot
interactive()
```

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at a Python tutorial (The Python Tutorial) to pick up the basics of Python if you are unfamiliar with the language, or you may learn enough Python as you go along with the examples in this tutorial. The latter strategy has proven to work for many newcomers to FEniCS[1]. Chapter 2.8.5 lists some good Python books.

The listed FEniCS program defines a finite element mesh, the discrete function spaces V_h and $\hat{V}_h$ over this mesh (i.e., the choice of elements), boundary conditions for u (i.e., the function u_0), $a(u,v)$, and $L(v)$. Thereafter, the unknown trial function u is computed. Then we can investigate u visually or analyze the computed values.

The first line in the program,

```
from dolfin import *
```

imports the key classes `UnitSquare`, `FunctionSpace`, `Function`, and so forth, from the DOLFIN library. All FEniCS programs for solving PDEs by the finite element method normally start with this line. DOLFIN is a software library with efficient and convenient C++ classes for finite element computing, and `dolfin` is a Python package providing access to this C++ library from Python programs. You can think of FEniCS is an umbrella, or project name, for a set of computational components, where DOLFIN is one important component for writing finite element programs. DOLFIN applies other components in the FEniCS suite under the hood, but newcomers to FEniCS programming do not need to care about this.

[1]The requirement of using Python and an abstract mathematical formulation of the finite element problem may seem difficult for those who are unfamiliar with these topics. However, the amount of mathematics and Python that is really demanded to get you productive with FEniCS is quited limited. And Python is an easy-to-learn language that you certainly will love and use far beyond FEniCS programming.

The statement

```
mesh = UnitSquare(6, 4)
```

defines a uniform finite element mesh over the unit square $[0, 1] \times [0, 1]$. The mesh consists of *cells*, which are triangles with straight sides. The parameters 6 and 4 tell that the square is first divided into $6 \cdot 4$ rectangles, and then each rectangle is divided into two triangles. The total number of triangles then becomes 48. The total number of vertices in this mesh is $7 \cdot 5 = 35$. DOLFIN offers some classes for creating meshes over very simple geometries. For domains of more complicated shape one needs to use a separate *preprocessor* program to create the mesh. The FEniCS program will then read the mesh from file.

Having a mesh, we can define a discrete function space `V` over this mesh:

```
V = FunctionSpace(mesh, 'CG', 1)
```

The second argument reflects the type of element, while the third argument is the degree of the basis functions on the element. Here, `'CG'` stands for Continuous Galerkin, implying the standard Lagrange family of elements. Insted of `'CG'` we could have written `'Lagrange'`. With degree 1, we simply get the standard linear Lagrange element, which is a triangle with nodes at the three vertices. Some finite element practitioners refer to this element as the "linear triangle". The computed u will be continuous and linearly varying in x and y over each cell in the mesh. Higher-order polynomial approximations over each cell are trivially obtained by increasing the third parameter to `FunctionSpace`.

In the mathematics, we distinguish between the trial and test spaces V_h and $\hat{V}_h$. The only difference in the present problem is the boundary conditions. In FEniCS we do not specify the boundary conditions as part of the function space, so it is sufficient to work with one common space `V` for the test and trial functions in the program:

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

The next step is to specify the boundary condition: $u = u_0$ on $\partial\Omega$. This is done by

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

where `u0` is an instance holding the u_0 values, and `u0_boundary` is an instance describing if a point lies on the boundary where u is specified. The term *instance* means a Python object of a particular type (such as `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangable terms. In other computer programming languages one may also use the term *variable* for the same thing. We shall in this tutorial mostly use the term *instance*, since that is most common in a Python context, but *object* will also be occasionally used where that is more natural.

Boundary conditions of the type $u = u_0$ are known as *Dirichlet conditions*, and also as *essential boundary conditions* in a finite element context. Naturally, the name of the DOLFIN class holding the information about Dirichlet boundary conditions is `DirichletBC`.

The `u0` variable refers to a `Function` instance, which is used to represent a mathematical function and/or a finite element function. A `Function` instance can be created in many ways, but the most straightforward recipe when we have a simple mathematical expression for u_0 is to write

```
u0 = Function(V, formula)
```

where `formula` is a string containing the mathematical expression written with C++ syntax (the formula is automatically turned into an efficient, compiled C++ function, see Chapter 2.8.6 for details on the syntax). The independent variables in the function expression are supposed to be available as a point vector `x`, where the first element `x[0]` corresponds to the x coordinate, the second element `x[1]` to the y coordinate, and (in a three-dimensional problem) `x[2]` to the z coordinate. With our choice of $u_0(x, y) = 1 + x^2 + 2y^2$, the formula string must be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

```
u0 = Function(V, '1 + x[0]*x[0] + 2*x[1]*x[1]')
```

The information about where to apply the `u0` function as boundary condition is coded in a method `inside` in a subclass of class `SubDomain`[2]:

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

on_boundary = Boundary()
```

The method `inside` shall return a boolean value: `True` if the point `x` lies on the Dirichlet boundary and `False` otherwise. The argument `on_boundary` is `True` if `x` is on the physical boundary of the mesh, so in the present case we can just return `on_boundary`. In later examples we will demonstrate how to set Dirichlet conditions on parts of the boundary, typically achieved by some test on the `x` values inside the `inside` method (as for the formula in `Function` instances, `x` in the `inside` method represents a point in space with coordinates `x[0]`, `x[1]`, etc.). The `inside` method is called for every discrete point in the mesh, which allows us to have boundaries where u are known also inside the domain, if desired. The choice of class name, here `Boundary`, is up to the programmer, but the class must be derived from `SubDomain` and it must have an `inside` method.

Newcomers to Python class programming often face some problems with understanding the `self` parameter in the `inside` function. For now it suffices to

[2]If you are unfamiliar with classes and class methods in Python, stay cool and just modify the many examples on boundary specifications found in this tutorial. It may well suffice to pick up Python class programming at a later stage.

know that `self` is a required first argument when defining a function in a class. There is no need to understand the `self` argument before in Chapter 2.7.2.

Before defining $a(u, v)$ and $L(v)$ we have to specify the f function:

```
f = Function(V, '-6')
```

When f is constant over the domain, `f` can be more efficiently represented as a `Constant` instance:

```
f = Constant(mesh, -6.0)
```

Now we have all the objects we need in order to specify this problem's $a(u, v)$ and $L(v)$:

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

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas (2.9)–(2.10)! This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify PDE problems with lots of PDEs and complicated terms in the equations. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

Having `a` and `L` defined, and information about essential (Dirichlet) boundary conditions in `bc`, we can formulate a `VariationalProblem`:

```
problem = VariationalProblem(a, L, bc)
```

Solving the variational problem for the solution `u` is just a matter of writing

```
u = problem.solve()
```

Unless otherwise stated, a sparse direct solver is used to solve the underlying linear system implied by the variational formulation. The type of sparse direct solver depends on which linear algebra package that is used by default. If DOLFIN is compiled with PETSc, that package is the default linear algebra backend, otherwise it is uBLAS. The FEniCS distribution for Ubuntu Linux contains PETSc, and then the default solver becomes the sparse LU solver from UMFPACK (which PETSc has an interface to). We shall later in Chapter 2.4 demonstrate how to get full control of the choice of solver and any solver parameters.

The `u` variable refers to a `Function` instance. Note that we first defined `u` as a `TrialFunction` and used it to specify `a`. Thereafter, we redefined `u` to be a `Function` representing the computed solution. This redefinition of the variable `u` is possible in Python and a programming practice in FEniCS applications.

The simplest way of quickly looking at `u` and the mesh is to say

```
plot(u)
plot(mesh)
interactive()
```

The `interactive()` call is necessary for the plot to remain on the screen. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine how the solution looks like.

It is also possible to dump the computed solution to file, e.g., in the VTK format:

```
file = File('poisson.pvd')
file << u
```

The `poisson.pvd` file can now be loaded into any front-end to VTK, say ParaView or VisIt. The `plot` function from Viper is intended for quick examination of the solution during program development. More in-depth visual investigations of finite element solution will normally benefit from using highly professional tools such as ParaView and VisIt.

2.1.4 Examining the Discrete Solution

We know that, in the particular boundary-value problem of Chapter 2.1.3, the computed solution u should equal the exact solution at the vertices of the cells. An important extension of our first program is therefore to examine the computed values of the solution, which is the focus of the present section.

A finite element function like u is expressed as a linear combination of basis functions ϕ_i (spanning the space V_h):

$$\sum_{j=1}^{N} U_j \phi_j \, . \tag{2.11}$$

By writing `u = problem.solve()` in the program, a linear system will be formed from a and L, and this system is solved for the $U_1, \ldots, U_N$ values. The $U_1, \ldots, U_N$ values are known as *degrees of freedom* of u. For Lagrange elements (and many other element types) U_k is simply the value of u at the node with global number k. (The nodes and cell vertices coincide for linear Lagrange elements, while for higher-order elements there may be additional nodes at the facets and in the interior of cells.)

Having `u` represented as a `Function` object, we can either evaluate `u(x)` at the nodes `x` in the mesh, or we can grab the values U_j directly by

```
u_nodal_values = u.vector()
```

The result is a DOLFIN `Vector` instance, which is basically an encapsulation of the vector object used in the linear algebra package that is applied to solve the linear system arising form the variational problem. Since we program in Python it is convenient to convert the `Vector` instance to a standard `numpy` array for further processing:

```
u_array = u_nodal_values.array()
```

With numpy arrays we can write “Matlab-like” code to analyze the data. Indexing is done with square brackets: `u_array[i]`, where the index `i` always starts at `0`.

The coordinates of the vertices in the mesh can be extracted by

```
coor = mesh.coordinates()
```

For a d-dimensional problem, `coor` is an $M \times d$ numpy array, M being the number of vertices in the mesh. Writing out the solution on the screen can now be done by a simple loop:

```
for i in range(len(u_array)):
    print 'u(%8g,%8g) = %g' % \
          (coor[i][0], coor[i][1], u_array[i])
```

The beginning of the output looks like

```
u(       0,       0) = 1
u(0.166667,       0) = 1.02778
u(0.333333,       0) = 1.11111
u(     0.5,       0) = 1.25
u(0.666667,       0) = 1.44444
u(0.833333,       0) = 1.69444
u(       1,       0) = 2
```

For Lagrange elements of degree higher than one, the vertices and the nodes do not coincide, and then the loop above is meaningless.

For verification purposes we want to compare the values of `u` at the nodes, i.e., the values of the vector `u_array`, with the exact solution given by `u0`. At each node, the difference between the computed and exact solution should be less than a small tolerance. The exact solution is given by the `Function` instance `u0`, which we can evaluate directly as `u0(coor[i])` at the vertex with global number `i`, or as `u0(x)` for any spatial point. Alternatively, we can make a finite element field `u_e`, representing the exact solution, whose values at the nodes are given by the `u0` function. With mathematics, $u_e = \sum_{j=1}^{N} E_j \phi_j$, where $E_j = u_0(x_j, y_j)$, (x_j, y_j) being the coordinates of node no. j. This process is known as interpolation. FEniCS has a function for performing the operation:

```
u_e = interpolate(u0, V)
```

The maximum error can now be computed as

```
u_e_array = u_e.vector().array()
diff = abs(u_array - u_e_array)
print 'Max error:', diff.max()

# or more compactly:
print 'Max error:', abs(u_e_array - u_array).max()
```

The value of the error should be at the level of the machine precision (10^{-16}).

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

```
# Compare numerical and exact solution at (0.5, 0.5)
center = (0.5, 0.5)
u_value = u(center)
u0_value = u0(center)
print 'numerical u at the center point:', u_value
print 'exact     u at the center point:', u0_value
```

Trying a 3×3 mesh, the output from the previous snippet becomes

```
numerical u at the center point: [ 1.83333333]
exact     u at the center point: [ 1.75]
```

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.

Mesh information can be gathered from the `mesh` instance, e.g., `mesh.num_cells()` returns the number of cells (triangles) in the mesh, `mesh.num_vertices()` returns the number of verticies in the mesh (with our choice of linear Lagrange elements this equals the number of nodes). The call `str(mesh)`, or simply writing `print mesh`, creates a short "pretty print" description of the mesh:

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

All mesh objects are of type `Mesh` so typing the command `pydoc dolfin.Mesh` in a terminal window will give a list of methods that can be called through any `Mesh` instance. In fact, `pydoc dolfin.X` shows the documentation of any DOLFIN name `X` (at the time of this writing, some names have missing or incomplete documentation).

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 snippet performs the task:

```
max_u = u_array.max()
u_array /= max_u
u.vector()[:] = u_array
print u.vector().array()
```

That is, we manipulate `u_array` as desired, and then we insert this array into `u`'s `Vector` instance. 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`. We can equally well insert the entries of `u_array` into `u`'s `numpy` array:

```
u.vector().array()[:] = u_array
```

All the code in this subsection can be found in the file `Poisson2D_D2.py`.

2.1.5 Formulating a Real Physical Problem

Perhaps you are not particularly amazed by viewing the simple surface of u in the test problem from Chapters 2.1.3 and 2.1.4. However, solving a real physical problem with a more interesting and amazing solution on the screen is only a matter of specifying a more exciting domain, boundary condition, and/or right-hand side f.

One possible physical problem regards the deflection of $D(x, y)$ of an elastic circular membrane with radius R, subject to a localized perpendicular pressure force, modeled as a Gaussian function. The appropriate PDE model is

$$-T\Delta D = p(x,y) \quad \text{in } \Omega = \{(x,y)\,|\,x^2+y^2 \leq R\}, \tag{2.12}$$

with

$$p(x,y) = \frac{A}{2\pi\sigma}\exp\left(-\frac{1}{2}\left(\frac{x-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y-y_0}{\sigma}\right)^2\right). \tag{2.13}$$

Here, T is the tension in the membrane (constant), p is the external pressure load, A the amplitude of the pressure, (x_0, y_0) the localization of the Gaussian pressure function, and σ the "width" of this function. The boundary condition is $D = 0$.

Introducing a scaling with R as characteristic length and $8\pi\sigma T/A$ as characteristic size of D, we can derive the equivalent scaled problem on the unit circle,

$$-\Delta w = 4\exp\left(-\frac{1}{2}\left(\frac{Rx-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{Ry-y_0}{\sigma}\right)^2\right), \tag{2.14}$$

with $w = 0$ on the boundary. We have that $D = Aw/(8\pi\sigma T)$.

A mesh over the unit circle can be created by

```
mesh = UnitCircle(n)
```

where `n` is the typical number of elements in the radial direction. You should now be able to figure out how to modify the `Poisson2D_D1.py` code to solve this membrane problem. More specifically, you are recommended to perform the following extensions:

1. initialize R, x_0, y_0, σ, T, and A in the beginning of the program,

2. build a string expression for p with correct C++ syntax (use printf formatting in Python to build the expression),

3. define the a and L variables in the variational problem for w and compute the solution,

4. plot the mesh, w, and the scaled pressure function p (the right-hand side of (2.14)),

5. write out the maximum real deflection D (i.e., the maximum of the w values times $A/(8\pi\sigma T)$).

Use variable names in the program similar to the mathematical symbols in this problem.

Choosing a small width σ (say 0.01) and a location (x_9, y_0) toward the circular boundary (say $(0.6R\cos\theta, 0.6R\sin\theta)$ for any $\theta \in [0, 2\pi]$), may produce an exciting visual comparison of w and p that demonstrates the very smoothed elastic response to a peak force (or mathematically, the smoothing properties of the Laplace operator). You need to experiment with the mesh resolution to get a smooth visual representation of p.

In the limit $\sigma \to \infty$, the right-hand side p of (2.14) approaches the constant 4, and then the solution should be $w(x, y) = 1 - x^2 - y^2$. Compute the absolute value of the difference between the exact and the numerical solution if $\sigma \geq 50$ and write out the maximum difference to provide some evidence that the implementation is correct.

You are strongly encouraged to spend some time on doing this exercise and play around with the plots and different mesh resolutions. A suggested solution to the exercise can be found in the file membrane1.py.

```
from dolfin import *

# Set pressure function:
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
#sigma = 50  # verification
pressure = '4*exp(-0.5*(pow((%g*x[0] - %g)/%g, 2)) '\
           '      - 0.5*(pow((%g*x[1] - %g)/%g, 2)))' % \
           (R, x0, sigma, R, y0, sigma)

n = 40   # approx no of elements in radial direction
mesh = UnitCircle(n)
V = FunctionSpace(mesh, 'CG', 1)

# Define boundary condition w=0

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

```
boundary = Boundary()
bc = DirichletBC(V, Constant(mesh, 0.0), boundary)

# Define variational problem
v = TestFunction(V)
w = TrialFunction(V)
p = Function(V, pressure)
a = dot(grad(v), grad(w))*dx
L = v*p*dx

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

# Plot solution and mesh
plot(mesh, title='Mesh over scaled domain')
plot(w, title='Scaled deflection')
plot(p, title='Scaled pressure')

# Find maximum real deflection
max_w = w.vector().array().max()
max_D = A*max_w/(8*pi*sigma*T)
print 'Maximum real deflection is', max_D

# Verification for "flat" pressure
if sigma >= 50:
    w_exact = Function(V, '1 - x[0]*x[0] - x[1]*x[1]')
    w_e = interpolate(w_exact, V)
    w_e_array = w_e.vector().array()
    w_array = w.vector().array()
    diff_array = abs(w_e_array - w_array)
    print 'Verification of the solution, max difference is %.4E' % \
          diff_array.max()

    difference = Function(V)
    difference.vector()[:] = diff_array
    #plot(difference, title='Error field for sigma=%g' % sigma)

# Should be at the end
interactive()
```

2.1.6 Computing Derivatives

In many Poisson and 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, `grad(u)` denotes the gradient. However, the gradient of a 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 $w = \nabla u$ by a finite element method[3], using the the same elements for the components of w as we used for u.

The variational problem for w reads: Find $w \in V_h$ such that

$$a(w, v) = L(v) \quad \forall v \in \hat{V}_h^{(g)}, \tag{2.15}$$

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{2.16}$$

$$L(v) = \int_\Omega v \cdot \nabla u \, \mathrm{d}x \,. \tag{2.17}$$

The function spaces V_h and $\hat{V}_h^{(g)}$ are vector versions of the function space for u, with boundary conditions removed (if V_h is the space we used for u, with no restrictions on boundary values, $V_h^{(g)} = \hat{V}_h^{(g)} = [V_h]^d$, 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 `gradu` in the code, is easy to solve in FEniCS:

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

a = dot(w, v)*dx
L = dot(grad(u), v)*dx
problem = VariationalProblem(a, L)
gradu = problem.solve()

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

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.

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

[3]This process is known as *projection*. 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 another function space with basis $\bar{\phi}_1, \ldots \bar{\phi}$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \bar{\phi}_j$, for suitable (new) coefficients $\bar{U}_j$.

```
gradu_x, gradu_y = gradu.split(deepcopy=True)  # extract components
plot(gradu_x, title='x-component of grad(u)')
plot(gradu_y, title='y-component of grad(u)')
```

The `gradu_x` and `gradu_y` variables behave as `Function` instances. In particular, we can extract the underlying arrays of nodal values by

```
gradu_x_array = gradu_x.vector().array()
gradu_y_array = gradu_y.vector().array()
```

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

```
gradu_array = gradu.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.

The program `Poisson2D_D3.py` extends the code `Poisson2D_D2.py` from Chapter 2.1.4 with computations and visualizations of the gradient. Examining the arrays `gradu_x_array` and `gradu_y_array`, or looking at the plots of `gradu_x` and `gradu_y`, quickly reveals that the computed `gradu` field does not equal the exact gradient $(2x, 4y)$ in this particular test problem where $u = 1 + x^2 + y^2$. 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). See Chapter 2.1.8 for illustrations of this phenomenon.

Representing the gradient by the same elements as we used for the solution is a very common step in finite element programs, so the formation and solution of a variational problem for w as shown above can be replaced by a one-line call:

```
gradu = project(grad(u), VectorFunctionSpace(mesh, 'CG', 1))
```

The `project` function can take an expression involving some finite element function in some space and project the expression onto another space. The applications 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 first-order field which is required by many visualization packages.

2.1.7 Computing Functionals

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

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u \cdot \nabla u \, \mathrm{d}x, \qquad (2.18)$$

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

$$||u - u_e|| = \left(\int_\Omega (u_e - u)^2 \,\mathrm{d}x \right)^{1/2}, \tag{2.19}$$

which is of particular interest when studying convergence properties. Sometimes the interst concerns the flux out of a part Γ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma p\nabla u \cdot \boldsymbol{n} \,\mathrm{d}s, \tag{2.20}$$

where $\boldsymbol{n}$ is an outward unit normal at Γ and p is a coefficient (see the problem in Chapter 2.1.12 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 (2.18) is described in the UFL language in the same manner as we describe weak forms:

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

The `assemble` call performs the integration. It is possible to restrict the integration to subdomains, using a mesh function to mark the subdomains as explained in Chapter 2.6.3. The program `membrane2.py` carries out the computation of the elastic energy $\frac{1}{2}||T\nabla w||^2$ in the membrane problem from Chapter 2.1.5.

Convergence Estimation. To illustrate error computations and convergence of finite element solutions, we modify the `Poisson2D_D3.py` program from Chapter 2.1.6 and specify a more complicated solution,

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

on the unit square. It follows that $u_0 = 0$ and that $f(x, y) = 2\omega^2\pi^2 u(x, y)$. We must define the appropriate boundary conditions, the exact solution, and the f function:

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

bc = DirichletBC(V, Constant(mesh, 0.0), Boundary())

omega = 1.0
u_exact = Function(V, 'sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                   {'omega': omega})

f = 2*pi**2*omega**2*u_exact
```

The computation of (2.19) can be done by

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

However, u_exact will in this expression be interpolated onto the function space V, i.e., 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. More accurate representation of the exact solution can be obtained by defining higher-order elements, say

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

The u function will here be automatically interpolated and represented in the Ve space. (When functions in different function spaces enter UFL expressions, they will be represented in the space of highest order before integrations are carried out.)

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 round-off 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. The usage is simple:

```
E = errornorm(u_exact, u, normtype='l2', degree=3)
```

Finally, we remove all plot calls and printouts of u values in the original program, and collect the computations in a function:

```
def compute(nx, ny, degree):
    mesh = UnitSquare(nx, ny)
    V = FunctionSpace(mesh, 'CG', degree)
    ...
    E = errornorm(u_exact, u, normtype='l2', degree=3)
    return E
```

Calling compute 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 divisions in x and y direction (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:

```
# Perform experiments
degree = int(sys.argv[1])
h = []  # element sizes
E = []  # errors
for nx in [4, 8, 16, 32, 64, 128]:
    h.append(1.0/nx)
    E.append(compute(nx, nx, degree))

# Convergence rates
from math import log as ln  # (log is a dolfin name too)
for i in range(1, len(E)):
    r = ln(E[i]/E[i-1])/ln(h[i]/h[i-1])
    print 'h=%10.2E r=%.2f' % (h[i], r)
```

The resulting program has the name `Poisson2D_D4.py`. Running this program for first-order elements yields the output

```
h=  1.25E-01 r=1.76
h=  6.25E-02 r=1.94
h=  3.12E-02 r=1.98
h=  1.56E-02 r=2.00
h=  7.81E-03 r=2.00
```

That is, we approach the expected second-order convergence of linear Lagrange elements as the meshes become sufficiently fine. Running the program for third-order elements results in the expected value $r = 4$:

```
h=  1.25E-01 r=4.09
h=  6.25E-02 r=4.03
h=  3.12E-02 r=4.01
h=  1.56E-02 r=4.00
h=  7.81E-03 r=4.00
```

Checking convergence rates is the next best method for verifying PDE codes (the best being exact recovery of a solution as in Chapter 2.1.4 and many other places in this tutorial).

Flux Functionals. To compute flux integrals like (2.20) we need to define the $\boldsymbol{n}$ vector, referred to as *facet normal* in FEniCS. If Γ 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)
```

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 Chapter 2.6.3. Assuming that the part corresponds to subdomain no. 0, the relevant form for the flux is `-p*dot(grad(u), n)*ds(0)`.

2.1.8 Quick Visualization with VTK

As we go along with examples it is fun to play around with `plot` commands and visualize what is computed. This section explains some useful visualization features.

The `plot(u)` command launches a FEniCS component called Viper, which applies the VTK package to visualize finite element functions. Viper is not a full-fledged, easy-to-use front-end to VTK (like ParaView or VisIt), but rather a thin layer on top of VTK's Python interface, allowing us to quickly visualize a DOLFIN function or mesh, or data in plain Numerical Python arrays, within a Python program. Viper is ideal for debugging, teaching, and initial scientific investigations. The visualization can be interactive, or you can steer and automate it through program statements. More advanced and professional visualizations are usually better done with advanced tools like ParaView, VisIt, or MayaVi2.

We have made a program `membrane1v.py` for the membrane deflection problem in Chapter 2.1.5 and added various demonstrations of Viper capabilities. You are encouraged to play around with `membrane1v.py` and modify the code as you read about various features. The `membrane1v.py` program solves the two-dimensional Poisson equation for a scalar field `w` (the membrane deflection).

The `plot` function can take additional arguments, such as a title of the plot, or a specification of a wireframe plot (elevated mesh) instead of a colored surface plot:

```
plot(mesh, title='Finite element mesh')
plot(w, wireframe=True, title='solution')
```

The three mouse buttons can be used to rotate, translate, and zoom the surface. Pressing `h` in the plot window makes a printout of several key bindings that are available in such windows. For example, pressing `m` in the mesh plot window dumps the plot of the mesh to an Encapsulated PostScript (`.eps`) file, while pressing `i` saves the plot in PNG format. All plotfile names are automatically generated as `simulationX.eps`, where `X` is a counter `0000`, `0001`, `0002`, etc., being increased every time a new plot file in that format is generated (the extension of PNG files is `.png` instead of `.eps`). Pressing `'o'` adds a red outline of a bounding box around the domain.

One can alternatively control the visualization from the program code directly. This is done through a `Viper` instance returned from the `plot` command. Let us grab this object and use it to 1) tilt the camera -65 degrees in latitude direction, 2) add some simple x and y axis, 3) change the default name of the plot files (generated by typing `m` and `i` in the plot window), 4) change the color scale, and 5) write the plot to a PNG and an EPS file. Here is the code:

```
viz1 = plot(w,
            wireframe=False,
            title='Scaled membrane deflection',
            rescale=False,
```

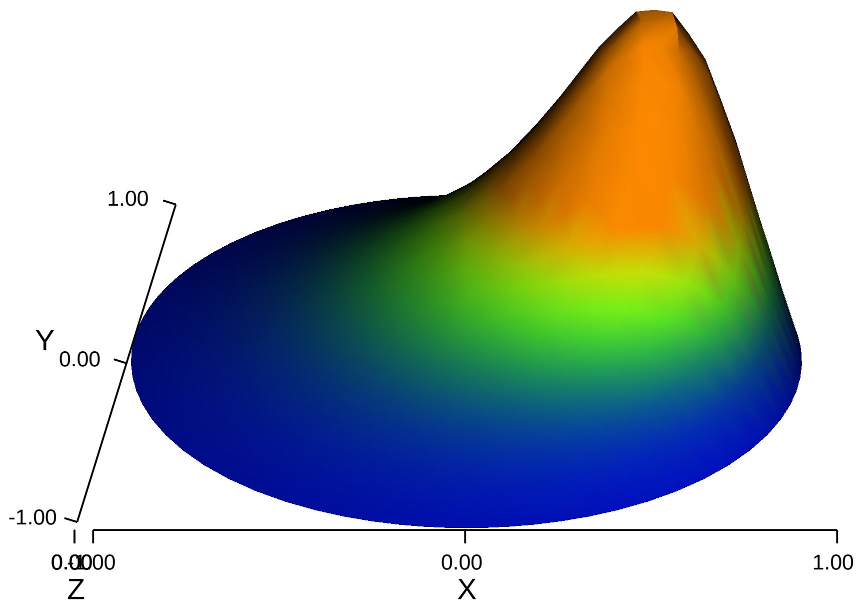


Figure 2.1: Plot of the deflection of a membrane.

```
               axes=True,               # include axes
               basename='deflection',   # default plotfile name
               )

viz1.elevate(-65) # tilt camera -65 degrees (latitude dir)
viz1.set_min_max(0, 0.5*max_w)  # color scale
viz1.update(w)    # bring settings above into action
viz1.write_png('deflection.png')
viz1.write_ps('deflection', format='eps')
```

The `format` argument in the latter line can also take the values `'ps'` for a standard PostScript file and `'pdf'` for a PDF file. Note the necessity of the `viz.update(w)` call – without it we will not see the effects of tilting the camera and changing the color scale. Figure 2.1 shows the resulting scalar surface.

2.1.9 Combining Dirichlet and Neumann Conditions

Let us make a slight extension of our two-dimensional Poisson problem from Chapter 2.1.1 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).

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

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, & (2.21)\\ u &= u_0 \text{ on } \Gamma_D, & (2.22)\\ -\frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N & (2.23)\end{aligned}$$

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

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

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

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

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 at the Γ_D. We have

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

and since $v = 0$ on Γ_D,

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

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

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x + \int_{\Gamma_N} gv \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x\,. \tag{2.24}$$

Expressing (2.24) in the standard notation $a(u, v) = L(v)$ is straightforward with

$$a(u,v) = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x, \tag{2.25}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s\,. \tag{2.26}$$

How does the Neumann condition impact the implementation? The code in the file `Poisson2D_D2.py` remains almost the same. Only two adjustments are necessary:

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

Step 1 can be coded as

```
class DirichletBoundary(SubDomain):
    def inside(self, 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

```
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (x[0] == 0 or x[0] == 1)
```

We remark that testing for an exact match of real numbers, as in `x[0] == 1`, is not good programming practice, because small round-off errors in the computation of the `x` values could make the outcome `False` even though `x` lies on the Dirichlet boundary. A better test is to check for equality with a tolerance:

```
class DirichletBoundary(SubDomain):
    def inside(self, 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 = Function(V, '-4*x[1]')
L = v*f*dx - v*g*ds
```

The `ds` variable implies a boundary integral, while `dx` implies an intergral over the domain Ω. No more modifications are necessary. Running the resulting program, found in the file `Poisson2D_DN1.py`, shows a successful verification – u equals the exact solution at all the nodes, regardless of how many elements we use.

2.1.10 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 at 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 Chapter 2.1.9 and define two separate functions for the two Dirichlet conditions:

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

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 $g = -4y$. For the left boundary Γ_0 we define the usual triple of a function for the boundary value, a subclass of `SubDomain` for defining the boundary of interest, and a `DirichletBC` instance:

```
u_L = Function(V, '1 + 2*x[1]*x[1]')

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

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

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

```
u_R = Function(V, '2 + 2*x[1]*x[1]')

class RightDirichletBoundary(SubDomain):
    def inside(self, 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, RightDirichletBoundary())
```

The various essential conditions are then collected in a list and passed onto our problem object of type `VariationalProblem`:

```
bc = [Gamma_0, Gamma_1]
...
problem = VariationalProblem(a, L, bc)
```

If the u values are constant at a part of the boundary, we may use a `Constant` instance instead of a full `Function` instance.

The file `Poisson2D_DN2.py` contains a complete program which demonstrates the constructions above. An extended example with multiple Neumann conditions would have been quite natural now, but this requires marking various parts of the boundary using the mesh function concept and is therefore left to Chapter 2.6.3.

2.1.11 A Linear Algebra Formulation

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

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

which is nothing but a linear system,

$$AU = b,$$

where the entries in A and b are given by

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

The examples so far have constructed a `VariationalProblem` instance and called its `solve` method to compute the solution `u`. The `VariationalProblem` instance creates a linear system $AU = b$ and calls an appropriate solution method for such systems. An alternative is dropping the use of a `VariationalProblem` instance and instead asking FEniCS to create the matrix A and right-hand side b, and then solve for the solution vector U of the linear system. The relevant statements read

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

The variables `a` and `L` are as before, i.e., `a` refers to the bilinear form involving a `TrialFunction` instance (say `u`) and a `TestFunction` instance (`v`), and `L` involves a `TestFunction` instance (`v`). From `a` and `L` the `assemble` function can compute the matrix elements $A_{i,j}$ and the vector elements b_i.

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the `bc.apply(A, b)` call performs the necessary modifications to the linear system. The first three statements above can alternatively be carried out by[4]

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

When we have multiple Dirichlet conditions stored in a list `bc`, as explained in Chapter 2.1.10, we must apply each condition in `bc` to the system:

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

Alternatively, we can make the call

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

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

```
if mesh.numCells() < 16:  # print for small meshes only
    print A.array()
    print b.array()
bc.apply(A, b)
if mesh.numCells() < 16:
    print A.array()
    print b.array()
```

You will see that `A` is modified in a symmetric way: for each degree of freedom that is known, the corresponding row and column is zero'ed out and 1 is placed on the main diagonal. The right-hand side `b` is modified accordingly (the column times the value of the degree of freedom is subtracted from `b`, and then the corresponding entry in `b` is replaced by the known value of the degree of freedom).

Sometimes it can be handy to transfer the linear system to Matlab or Octave for futher analysis, e.g., computation of eigenvalues of A. This is easily done by opening a `File` instance with a filename extension `.m` and dump the `Matrix` and `Vector` instances as follows:

[4]The essential boundary conditions are now applied to the element matrices and vectors prior to assembly.

```
mfile = File('A.m'); mfile << A
mfile = File('b.m'); mfile << b
```

The data files `A.m` and `b.m` can be loaded directly into Matlab or Octave.

The complete code where our Poisson problem is solved by forming the linear system $AU = b$ explicitly, is stored in the files `Poisson2D_DN_la1.py` (one common Dirichlet condition) and `Poisson2D_DN_la2.py` (two separate Dirichlet conditions).

Creating the linear system explicitly in the user's program, as an alternative to using a `VariationalProblem` instance, 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. Chapters 2.3.2 and 2.3.3 deal with this topic in detail.

2.1.12 A Variable-Coefficient Poisson Problem

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

$$\begin{aligned} -\nabla\cdot[p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \quad \text{on } \partial\Omega\,. \end{aligned} \tag{2.27}$$

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

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

What are the modifications we need to do in the `Poisson2D_D2.py` program from Chapter 2.1.4?

1. `f` must be a `Function` since it is no longer a constant,
2. a new `Function` `p` must be defined for the variable coefficient,
3. the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE in (2.27) and integrating by parts now results in

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

The function spaces for u and v are the same as in Chapter 2.1.2, 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

$$a(u,v) = \int_\Omega p\nabla v\cdot\nabla u\,\mathrm{d}x, \tag{2.28}$$

$$L(v) = \int_\Omega vf\,\mathrm{d}x\,. \tag{2.29}$$

In the code from Chapter 2.1.3 we must replace

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

by

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

The definitions of `p` and `f` read

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

No additional modifications are necessary. The complete code can be found in in the file `Poisson2D_Dvc.py`. You can run it and confirm that it recovers the exact u at the nodes.

The flux $-p\nabla u$ may be of particular interest in variable-coefficient Poisson problems. As explained in Chapter 2.1.6, 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 $w\in V_h^{(g)}$ such that

$$a(w,v)=L(v^g)\quad \forall v\in\hat{V}_h^{(g)}, \tag{2.30}$$

where

$$a(w,v) = \int_\Omega w\cdot v\,\mathrm{d}x, \tag{2.31}$$

$$L(v^g) = \int_\Omega v\cdot(-p\nabla u)\,\mathrm{d}x\,. \tag{2.32}$$

This problem is identical to the one in Chapter 2.1.6, except that p enters the integral in L.

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

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

a = dot(w, v)*dx
L = dot(-p*grad(u), v)*dx
problem = VariationalProblem(a, L)
flux = problem.solve()
```

The convenience function `project` was made to condense the frequently occuring statements above:

```
flux = project(-p*grad(u),
               VectorFunctionSpace(mesh, 'CG', 1))
```

Plotting the flux vector field is naturally as easy as plotting the gradient in Chapter 2.1.6:

```
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))')
```

Data analysis of the nodal values of the flux field may conveniently apply the underlying `numpy` arrays:

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

The program `Poisson2D_Dvc.py` contains in addition some plots, including a curve plot comparing `flux_x` and the exact counterpart along the line $y = 1/2$. The associated programming details related to this visualization are explained in Chapter 2.1.13.

2.1.13 Visualization 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 tools for structured data, like the data appearing in finite difference simulations and image analysis. We shall demonstrate the potential of such tools.

A necessary first step is to transform our `mesh` instance to an object representing a rectangle with equally-shaped *rectangular* cells. The Python package `scitools` has this type of structure, called a `UniformBoxGrid`. 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 grid. In such grids, 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 straiightforward, yet it frequently leads to obscure indexing errors. The `BoxField` object in `scitools` takes conveniently care of the details of the transformation. With a `BoxField` defined on a `UniformBoxGrid` it is very easy to call up more standard plotting packages to visualize the solution along lines in the domain or as 2D contours or lifted surfaces.

Let us go back to the `Poisson2D_Dvc.py` code from Chapter 2.1.12 and map `u` onto a `BoxField` instance:

```
from scitools.BoxField import *
u_box = dolfin_function2BoxField(u, mesh, (nx,ny), uniform_mesh=True)
```

Here, `nx` and `ny` are the number of divisions in each space direction that were used when calling `UnitSquare` to make the `mesh` instance. The result `u_box` is a `BoxField` instance that supports "finite difference" indexing and an underlying grid suitable for `numpy` operations on 2D data. Also 1D and 3D functions in DOLFIN can be turned into `BoxField` instances.

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. Here is an example of writing out the coordinates and the field value at a grid point with indices `i` and `j` (going from 0 to `nx` and `ny`, respectively, from lower left to upper right corner):

```
i = nx; j = ny   # upper right corner
print 'u(%g,%g)=%g' % (u_box.grid.coor[X][i],
                       u_box.grid.coor[Y][j],
                       u_box.values[i,j])
```

For instance, the x coordinates are reached by `u_box.grid.coor[X]`, where `X` is an integer (0) imported from `scitools.BoxField`. The `grid` attribute is an instance of class `UniformBoxGrid`.

Many plotting programs can be used to visualize the data in `u_box`. Matplotlib is now a very popular plotting program in the Python world and could be used to make contour plots of `u_box`. However, other programs like Gnuplot, VTK, and Matlab have better support for surface plots. Our choice in this tutorial is to use the Python package `scitools.easyviz`, which offers a uniform Matlab-like syntax to various plotting packages such as Gnuplot, Matplotlib, VTK, OpenDX, Matlab, and others. With `scitools.easyviz` we write one set of statements, close to what one would do in Matlab or Octave, and then it is easy to switch between different plotting programs, at a later stage, through a command-line option, a line in a configuration file, or an import statement in the program. By default, `scitools.easyviz` employs Gnuplot as plotting program, and this is a highly relevant choice for scalar fields over two-dimensional, structured meshes, or for curve plots along lines through the domain.

A contour plot is made by the following `scitools.easyviz` command:

```
from scitools.easyviz import contour, title, hardcopy
contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        5, clabels='on')
title('Contour plot of u')
hardcopy('u_contours.eps')

# or more compact syntax:
contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        5, clabels='on',
        hardcopy='u_contours.eps', title='Contour plot of u')
```

The resulting plot can be viewed in Figure 2.3a. The `contour` function needs arrays with the x and y coordinates expanded to 2D arrays (in the same way as demanded when making vectorized numpy calculations of arithmetic expressions over all grid points). The correctly expanded arrays are stored in grid.coorv. The above call to contour creates 5 equally spaced contour lines, and with clabels='on' the contour values can be seen in the plot.

Other functions for visualizing 2D scalar fields are surf and mesh as known from Matlab. Because the from dolfin import * statement imports several names that are also present in scitools.easyviz (e.g., plot, mesh, and figure), we use functions from the latter package through a module prefix ev (for easyviz) from now on:

```
import scitools.easyviz as ev
ev.figure()
ev.surf(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        shading='interp', colorbar='on',
        title='surf plot of u', hardcopy='u_surf.eps')

ev.figure()
ev.mesh(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        title='mesh plot of u', hardcopy='u_mesh.eps')
```

Figure 2.2 exemplifies the surfaces arising from the two plotting commands above. You can type pydoc scitools.easyviz in a terminal window to get a full tutorial.

A handy feature of BoxField is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearst grid line. In 3D fields one can also extract data in a plane. Say we want to plot u along the line $y = 1/2$ in the grid. The grid points, x, and the u values along this line, uval, are extracted by

```
start = (0, 0.5)
x, uval, 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. Plotting u versus the x coordinate along this line, using scitools.easyviz, is now a matter of

```
ev.figure()  # new plot window
ev.plot(x, uval, 'r-')  # 'r--: red solid line
ev.title('Solution')
ev.legend('finite element solution')

# or more compactly:
ev.plot(x, uval, 'r-', title='Solution',
        legend='finite element solution')
```

A more exciting plot compares the projected numerical flux in x direction along the line $y = 1/2$ with the exact flux:

```
ev.figure()
flux_x_box = dolfin_function2BoxField(flux_x, mesh, (nx,ny),
                                      uniform_mesh=True)
x, fluxval, y_fixed, snapped = \
      flux_x_box.gridline(start, direction=X)
y = y_fixed
flux_x_exact = -(x + y)*2*x
ev.plot(x, fluxval, 'r-',
        x, flux_x_exact, 'b-',
        legend=('numerical (projected) flux', 'exact flux'),
        title='Flux in x-direction (at y=%g)' % y_fixed,
        hardcopy='flux.eps')
```

As seen from Figure 2.3b, the numerical flux is accurate except in the elements closest to the boundaries.

It should be easy with the information above to a transform finite element field over a uniform rectangular or box-shaped mesh to the corresponding `BoxField` instance and perform Matlab-style visualizations of the whole field or the field over planes or along lines through the domain. By the transformation to a regular grid we have some more flexibility than what Viper offers. (It should be added that comprehensive tools like VisIt, MayaVi2, or ParaView also have the possibility for plotting fields along lines and extracting planes in 3D geometries, though usually with less degree of control compared to Gnuplot, Matlab, and Matplotlib.)

2.1.14 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. The relevant technicalities are therefore explained below.

Consider the simple problem

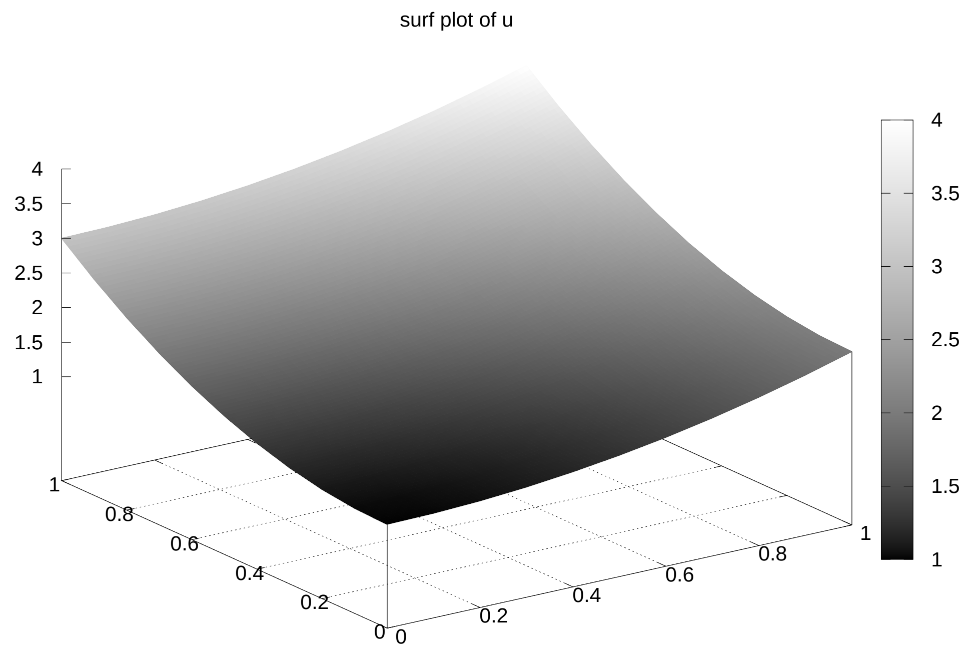
$$u''(x) = 2 \text{ in } [0,1], \quad u(0) = 0,\ u(1) = 0, \tag{2.33}$$

with exact solution $u(x) = x^2$. Our aim is to formulate and solve this problem in a 2D and a 3D domain as well. We may generalize the domain $[0,1]$ to a box of any size in the y and z directions and pose homogeneous Neumann conditions $\partial u/\partial n = 0$ at all additional boundaries $y =$ const and $z =$ const to ensure that u only varies with x. For example, let us choose a unit hypercube as domain: $\Omega = [0,1]^d$, where d is the number of space dimensions. The generalized d-dimensional Poisson problem then reads

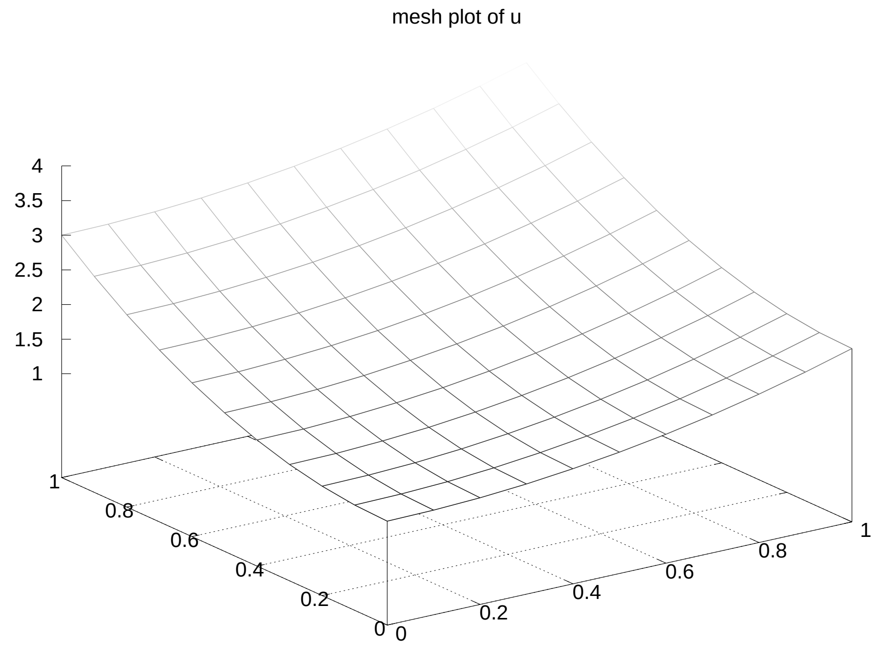
$$\begin{array}{rcl} \Delta u &=& 2 \quad \text{in } \Omega, \\ u &=& 0 \quad \text{on } \Gamma_0, \\ u &=& 1 \quad \text{on } \Gamma_1, \\ \frac{\partial u}{\partial n} &=& 0 \quad \text{on } \partial\Omega \cap (\Gamma_0 \cup \Gamma_1)\,, \end{array} \tag{2.34}$$

where Γ_0 is the side of the hypercube where $x = 0$, and where Γ_1 is the side where $x = 1$.

Implementing (2.34) for any d is no more complicated than solving a dimension-specific problem. The only non-trivial part of the code is actually to define the

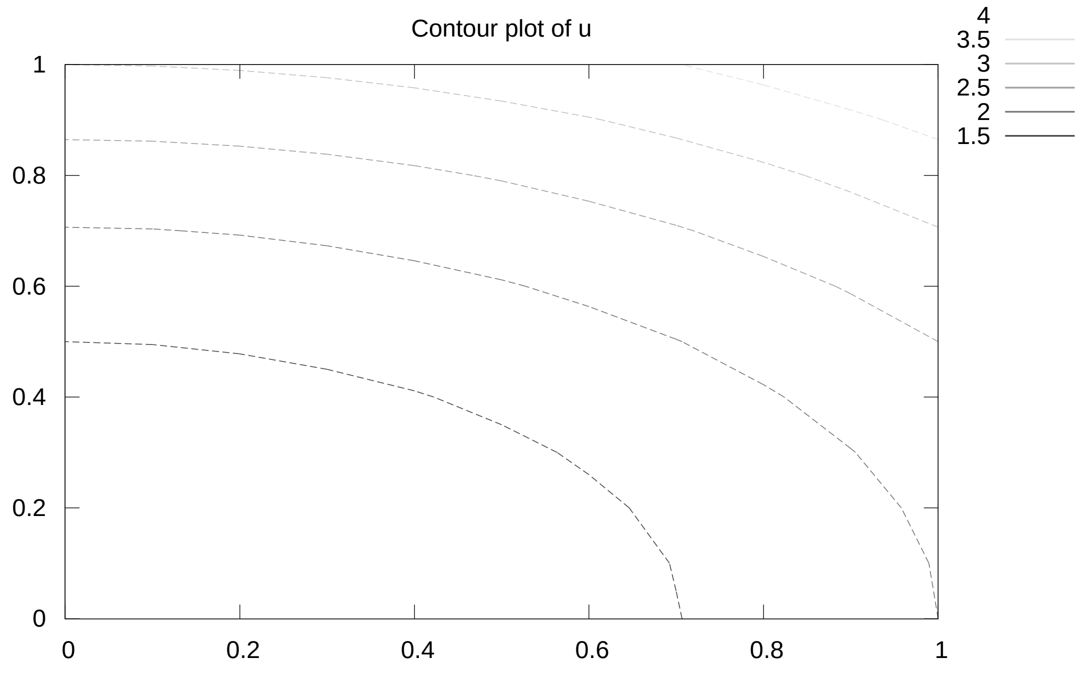


(a)

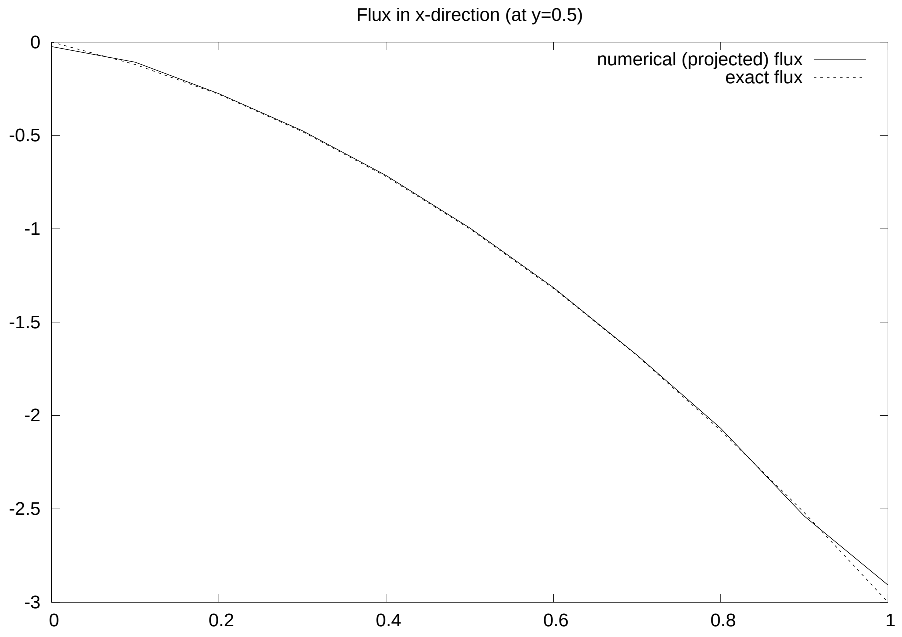


(b)

Figure 2.2: Examples on plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) a surface plot of the solution; (b) lifted mesh plot of the solution.



(a)



(b)

Figure 2.3: Examples on plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) contour plot of the solution; (b) curve plot of the exact flux $-p\partial u/\partial x$ against the corresponding projected numerical flux.

mesh. We use the command line to provide user-input to the program. The first argument can be the degree of the polynomial in the finite element basis functions. Thereafter, we supply the cell divisions in the various spatial directions. The number of command-line arguments will then imply the number of space dimensions. For example, writing `3 10 3 4` on the command-line means that we want to approximate u by piecewise polynomials of degree 3, and that the domain is a three-dimensional cube with $10 \times 3 \times 4$ divisions in the x, y, and z directions, respectively. Each of the $10 \times 3 \times 4 = 120$ boxes will be divided into six tetrahedras. The Python code can be quite compact:

```
degree = int(sys.argv[1])
divisions = [int(arg) for arg in sys.argv[2:]]
d = len(divisions)
domain_type = [UnitInterval, UnitSquare, UnitCube]
mesh = domain_type[d-1](*divisions)
V = FunctionSpace(mesh, 'CG', degree)
```

First note that although `sys.argv[2:]` holds the divisions of the mesh, all elements of the list `sys.argv[2:]` are string objects, so we need to explicitly convert each element to an integer. The construction `domain_type[d-1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends each component of the list `divisions` as a separate argument. For example, in a 2D problem where `divisions` has two elements, the statement

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

is equivalent to

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

The next part of the program is to set up the boundary conditions. Since the Neumann conditions have $\partial u/\partial n = 0$ we can omit the boundary integral from the weak form. We then only need to take care of Dirichlet conditions at two sides:

```
tol = 1E-14   # tolerance for coordinate comparisons
class DirichletBoundary0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0]) < tol

class DirichletBoundary1(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0] - 1) < tol

bc0 = DirichletBC(V, Constant(mesh, 0), DirichletBoundary0())
bc1 = DirichletBC(V, Constant(mesh, 1), DirichletBoundary1())
bc = [bc0, bc1]
```

Note that this code is independent of the number of space dimensions. So are the statements defining and solving the variational problem:

```
v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, -2)
a = dot(grad(v), grad(u))*dx
L = v*f*dx

problem = VariationalProblem(a, L, bc)
u = problem.solve()
```

The complete code is found in `Poisson123D_DN1.py`.

Observe that if we actually want to test variations in one selected space direction, parameterized by `e`, we only need to replace `x[0]` in the code by `x[e]` (!). The parameter `e` could be given as the second command-line argument. This extension appears in the file `Poisson123D_DN2.py`. You can run a 3D problem with this code where u varies in, e.g., z direction and is approximated by, e.g., a 5-th degree polynomial. For any legal input the numerical solution coincides with the exact solution at the nodes (because the exact solution is a second-order polynomial).

2.2 Nonlinear Problems

Now we shall address how to solve nonlinear PDEs in FEniCS. Our sample PDE for implementation is taken as a nonlinear Poisson equation:

$$-\nabla \cdot (q(u)\nabla u) = f\,. \tag{2.35}$$

The coefficient $q(u)$ makes the equation nonlinear (unless $q(u)$ is a constant).

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 is 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

$$u(x_0,\ldots,x_d) = \left((2^{m+1}-1)x_0 + 1\right)^{1/(m+1)} - 1\,. \tag{2.36}$$

The variational formulation of our model problem reads: Find $u \in V$ such that

$$F(u;v) = 0 \quad \forall v \in \hat{V}, \tag{2.37}$$

where

$$F(u;v) = \int_\Omega \nabla v \cdot (q(u)\nabla u)\;\mathrm{d}x, \tag{2.38}$$

and

$$\begin{aligned}
\hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } x_0 = 1\},\\
V &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } v = 1 \text{ on } x_0 = 1\}.
\end{aligned}$$

The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces: Find $u_h \in V_h$ such that

$$F(u_h; v) = 0 \quad \forall v \in \hat{V}_h, \tag{2.39}$$

with $u_h = \sum_{j=1}^{N} U_j \phi_j$. Since F is a nonlinear function of u_h, (2.39) gives rise to a system of nonlinear algebraic equations. From now on the interest is only in the discrete problem, and as mentioned in Chapter 2.1.2, we simply write u instead of u_h to get a closer notation between the mathematics and the Python code. When the exact solution needs to be distinguished, we denote it by u_e.

FEniCS can be used in alternative ways for solving a nonlinear PDE problem. We shall in the following subsections go through four solution strategies: 1) a simple Picard-type iteration, 2) a Newton method at the algebraic level, 3) a Newton method at the PDE level, and 4) an automatic approach where FEniCS attacks the nonlinear variational problem directly. The "black box" strategy 4) is definitely the simplest one from a programmer's point of view, but the others give more control of the solution process for nonlinear equations (which also has some pedagogical advantages).

2.2.1 Picard Iteration

Picard iteration is an easy way of handling nonlinear PDEs: we simply use a known, previous solutions in the nonlinear terms such that these terms become linear in the unknown u. For our particular problem, this means that 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*

$$\nabla \cdot \left(q(u^k)\nabla u^{k+1}\right) = 0, \quad k = 0, 1, \ldots \tag{2.40}$$

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 Equation (2.40). Equivalently, we can use u^k in $q(u)$ in the nonlinear variational problem (2.38) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u \in V$ such that

$$F(u^{k+1}; v) = 0 \quad \forall v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{2.41}$$

with

$$F(u^{k+1}; v) = \int_\Omega \nabla v \cdot \left(q(u^k)\nabla u^{k+1}\right) \,\mathrm{d}x\,. \tag{2.42}$$

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

$$a(u^{k+1}, v) = L(v), \tag{2.43}$$

with

$$a(u,v) = \int_\Omega \nabla v \cdot \left(q(u^k)\nabla u\right)\,\mathrm{d}x \qquad (2.44)$$

$$L(v) = 0\,. \qquad (2.45)$$

The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| <$ tol, where tol is small, 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+}, called uk and u in the code below. The algorithm can then be expressed as follows:

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

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
uk = Function(V, '0.0')  # previous (known) u
a = dot(grad(v), q(uk)*grad(u))*dx
f = Constant(mesh, 0.0)
L = f*v*dx

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

Note that we use numpy functionality to compute the norm of the difference between the two most recent solutions. Here we apply the maximum/infinity 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 nlPoisson_Picard.py contains the complete code for this problem. The implementation is d dimensional, with mesh construction and setting of Dirichlet conditions as explained in Chapter 2.1.14. For a 33×33 grid with $m = 2$ we need 9 iterations for convergence when the tolerance is 10^{-5}.

2.2.2 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 (2.38), the discrete version (2.39) 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 (2.39)):

$$F_i(U_1, \ldots, U_N) \equiv \sum_{j=1}^N \int_\Omega \nabla\hat{\phi}_i \cdot \left(q(\sum_{\ell=1}^N U_\ell \phi_\ell) \nabla\phi_j U_j \right) \,\mathrm{d}x = 0, \quad i = 1, \ldots, N. \quad (2.46)$$

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

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k, \ldots, U_N^k)\delta U_j = -F_i(U_1^k, \ldots, U_N^k), \quad (2.47)$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \quad (2.48)$$

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.

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

$$\int\limits_\Omega \left[\nabla\hat{\phi}_i \cdot ((q'(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^N U_j^k \phi_j)) + \nabla\hat{\phi}_i \cdot (q(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\nabla\phi_j) \right] \,\mathrm{d}x\,. \quad (2.49)$$

The following results were used to obtain (2.49):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j}\sum_{j=1}^N U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j} q(u) = q'(u)\phi_j\,. \quad (2.50)$$

We can reformulate the Jacobian matrix in (2.49) by introducing the short notation $u^k = \sum_{j=1}^N U_j^k \phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[\nabla\hat{\phi}_i \cdot \left(q'(u^k)\phi_j \nabla u^k \right) + \nabla\hat{\phi}_i \cdot \left(q(u^k)\nabla\phi_j \right) \right] \,\mathrm{d}x\,. \quad (2.51)$$

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 discrete weak form

$$\int_\Omega \left[\nabla v \cdot \left(q'(u^k)\delta u \nabla u^k \right) + \nabla v \cdot \left(q(u^k)\nabla\delta u \right) \right] \,\mathrm{d}x = -\int_\Omega \nabla v \cdot \left(q(u^k)\nabla u^k \right) \,\mathrm{d}x\,. \quad (2.52)$$

Equation (2.52) fits the standard form $a(u,v) = L(v)$ with

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

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 $\Delta u^0 = 0$. The receipe for solving this problem appears in Chapters 2.1.2, 2.1.3, and 2.1.9. The code for computing u^0 becomes as follows:

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

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

Gamma_0  = DirichletBC(V, Constant(mesh, 0.0),
                       LeftDirichletBoundary())
Gamma_1 = DirichletBC(V, Constant(mesh, 1.0),
                      RightDirichletBoundary())
bc_u = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, i.e., m=0)
v = TestFunction(V)
u = TrialFunction(V)
a = dot(grad(v), grad(u))*dx
f = Constant(mesh, 0.0)
L = v*f*dx
A, b = assemble_system(a, L, bc_u)
uk = Function(V)
solve(A, uk.vector(), b)
```

Here, `uk` denotes the solution function for the previous iteration, so that solution after each Newton iteration is `u = uk + omega*du`. Initially, `uk` is the initial guess we call u^0 in the mathematics.

The Dirichlet boundary conditions for 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 Dirichlet values. We therefore define an additional list of Dirichlet boundary conditions instances for δu:

```
Gamma_0_du  = DirichletBC(V, Constant(mesh, 0.0),
                          LeftDirichletBoundary())
```

```
Gamma_1_du = DirichletBC(V, Constant(mesh, 0.0),
                         RightDirichletBoundary())
bc_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 = uk + omega*du
a = dot(grad(v), q(uk)*grad(du))*dx + \
    dot(grad(v), Dq(uk)*du*grad(uk))*dx
L = -dot(grad(v), q(uk)*grad(uk))*dx
```

The Newton iteration loop is very similar to the Picard iteration loop in Chapter 2.2.1:

```
du = Function(V)
u  = Function(V)  # u = uk + 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, bc_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print 'Norm:', eps
    u.vector()[:] = uk.vector() + omega*du.vector()
    uk.assign(u)
```

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

```
u.assign(uk)  # u = uk
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` instance. 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 Chapter 2.1.14. The complete program appears in the file `nlPoisson_algNewton.py`.

2.2.3 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 of Chapter 2.2.2.

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

$$u^{k+1} = u^k + \delta u \tag{2.53}$$

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

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{2.54}$$

and dropping other 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,

$$\nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = -\nabla\cdot\left(q(u^k)\nabla u^k\right), \tag{2.55}$$

The weak form of this PDE is derived by multiplying by a test function v and integrating over Ω, integrating the second-order derivatives by parts:

$$\int_\Omega \left(\nabla v\cdot\left(q(u^k)\nabla\delta u\right) + \nabla v\cdot\left(q'(u^k)\delta u\nabla u^k\right)\right)\,\mathrm{d}x = \int_\Omega \nabla v\cdot\left(q(u^k)\nabla u^k\right)\,\mathrm{d}x\,. \tag{2.56}$$

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

$$a(\delta u, v) = \int_\Omega \left(\nabla v\cdot\left(q(u^k)\nabla\delta u\right) + \nabla v\cdot\left(q'(u^k)\nabla u^k\right)\right)\,\mathrm{d}x, \tag{2.57}$$

$$L(v) = \int_\Omega \nabla v\cdot\left(q(u^k)\nabla u^k\right)\,\mathrm{d}x\,. \tag{2.58}$$

The continuous function spaces V and $\hat{V}$, and their discrete counterparts, V_h and $\hat{V}_h$, are as in the linear Poisson problem from Chapter 2.1.2.

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 $a_0(u, v) = \int_\Omega \nabla v\cdot\nabla u\,\mathrm{d}x$ and $L(v) = 0$. Thereafter, we enter a loop and solve $a(\delta u, v) = L(v)$ for δt and compute a new approximation $u^{k+1} = u^k + \delta u$. Looking at (2.58) and (2.58), we see that the variational form is the same as for the Newton method at the algebraic level in Chapter 2.2.2. 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 is more straightforward. 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 Chapter 2.2.2 and is found in the file `nlPoisson_pdeNewton.py` (for the fun of it we use a `VariationalProblem` instance instead of assembling a matrix and vector and calling `solve`). 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 (Chapter 2.2.2).

2.2.4 Solving the Nonlinear Variational Problem Directly

DOLFIN has a built-in Newton solver and is able to automate the computation of nonlinear, stationary boundary-value problems. The automation is demonstrated next. A nonlinear variational problem (2.37) can be solved by

```
VariationalProblem(a, L, nonlinear=True)
```

where `L` corresponds to the form $F(u;v)$ in (2.37) and `a` is a form for the derivative of `L`.

The `L` form is straightforwardly defined (assuming `q(u)` is coded):

```
v = TestFunction(V)
u = Function(V)  # the unknown
L = dot(grad(v), q(u)*grad(u))*dx
```

The derivative `a` of `L` is formally the Gateaux derivative of $F(u;v)$ in the direction of the trial function. Technically, this Gateaux derivative is derived by computing

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u^k+\epsilon\delta u; v) \qquad (2.59)$$

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

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

and obtain

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

which leads to

$$\int_\Omega \nabla v\cdot\left[q'(u^k)\delta u\nabla(u^k)+q(u^k)\nabla\delta u\right]\,\mathrm{d}x, \qquad (2.60)$$

as $\epsilon \to 0$. This last expression is the Gateaux derivative of F and is denoted by $a(\delta u, v)$. The corresponding implementation goes as

```
du = TrialFunction(V)
a = dot(grad(v), q(u)*grad(du))*dx + \
    dot(grad(v), Dq(u)*du*grad(u))*dx
```

The UFL language we use to specify weak forms supports differentiation of forms. This means that when `L` is given as above, we can simply compute the Gateaux derivative by

```
a = derivative(L, u, du)
```

The differentiation is done symbolically so no numerical approximation formulas are involved. The `derivative` function is obviously very convenient in problems where differentiating `L` by hand implies lengthy calculations.

The solution of the nonlinear problem is now a question of two statements:

```
problem = VariationalProblem(a, L, bc, nonlinear=True)
u = problem.solve(u)
```

The `u` we feed to `problem.solve` is filled with the solution and returned, implying that the `u` on the left-hand side actually refers to the same `u` as provided on the right-hand side[5]. The file `nlPoisson_vp1.py` contains the complete code, where `a` is calculated manually, while `nlPoisson_vp2.py` is a counterpart where `a` is computed by `derivative(L, u, du)`. The latter file represents clearly the most automated way of solving the present nonlinear problem in FEniCS.

2.3 Time-Dependent Problems

The examples in Chapter 2.1 illustrate that solving linear, stationary PDE problems with the aid of FEniCS is easy and requires little programming. That is, FEniCS automates the spatial discretization by the finite element method. The solution of nonlinear problems, as we showed in Chapter 2.37, can also be automated (cf. Chapter 2.2.4), but many scientists will prefer to code the solution strategy of the nonlinear problem themselves and experiment with various combination of strategies in difficult problems. Time-dependent problems are

[5]Python has a convention that all input data to a function or class method are represented as arguments, while all output data are returned to the calling code. Data used as both input and output, as in this case, will then be arguments and returned. It is not necessary to have a variable on the left-hand side, as the function instance is modified correctly anyway, but it is convention that we follow here.

somewhat similar in this respect: we have to add a time discretization scheme, which is often quite simple, making it natural to explicitly code the details of the scheme so that the programmer have full control. We shall explain how easily this is accomplished through examples.

2.3.1 A Diffusion Problem and Its Discretization

Our time-dependent model problem for teaching purposes is naturally the simplest extension of the Poisson problem into the time domain, i.e., the diffusion problem

$$\begin{aligned}\frac{\partial u}{\partial t} &= \Delta u + f \text{ in } \Omega, && (2.61)\\ u &= u_0 \text{ on } \partial\Omega, && (2.62)\\ u &= I \text{ for } t = 0\,. && (2.63)\end{aligned}$$

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

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

Let superscript k denote a quantity at time t_k, where k is an integer counting time levels. For example, u^k means u at time level k. A finite difference discretization in time first consists in sampling the PDE at some time level, say k:

$$\frac{\partial}{\partial t}u^k = \Delta u^k + f^k\,. \qquad (2.64)$$

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

$$\frac{\partial}{\partial t}u^k \approx \frac{u^k - u^{k-1}}{\Delta t}, \qquad (2.65)$$

where Δt is the time discretization parameter. Inserting (2.65) in (2.64) yields

$$\frac{u^k - u^{k-1}}{\Delta t} = \Delta u^k + f^k\,. \qquad (2.66)$$

This is our time-discrete version of the diffusion PDE (2.61). Reordering (2.66) so that u^k appears on the left-hand side only, shows that (2.66) is a recursive set of spatial (stationary) problems for u^k (assuming u^{k-1} is know from compuations at the previous time level):

$$\begin{aligned}u^0 &= I, && (2.67)\\ u^k - \Delta u^k &= u^{k-1} + \Delta t f^k, \quad k = 1, 2, \ldots && (2.68)\end{aligned}$$

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

We use a finite element method to solve the equations (2.67) and (2.68). This requires turning the equations into weak forms. As usual, we multiply by a test function $v \in \hat{V}$ and integrate second-derivatives by parts. Introducing the symbol u for u^k (which is natural in the program too), the resulting weak forms can be conveniently written in the standard notation: $a_0(u,v) = L_0(v)$ for (2.67) and $a(u,v) = L(v)$ for (2.68), where

$$a_0(u,v) = \int_\Omega vu \,\mathrm{d}x, \tag{2.69}$$

$$L_0(v) = \int_\Omega vI \,\mathrm{d}x, \tag{2.70}$$

$$a(u,v) = \int_\Omega \left(vu + \Delta t \nabla v \cdot \nabla u\right) \mathrm{d}x, \tag{2.71}$$

$$L(v) = \int_\Omega v\left(u^{k-1} + \Delta t f^k\right) \mathrm{d}x\,. \tag{2.72}$$

The continuous variational problem is to find $u^0 \in V$ such that $a_0(u^0,v) = L_0(v)$ holds for all $v \in \hat{V}$, and then find $u^k \in V$ such that $a(u^k,v) = L(v)$ for all $v \in \hat{V}$, $k = 1, 2, \ldots$.

Approximate solutions in space are found by restricting the functional spaces V and $\hat{V}$ to finite-dimensional spaces V_h and $\hat{V}_h$, exactly as we have done in the Poisson problems. We shall use the symbol u for the finite element approximation at time t_k. In case we need to distinguish this space-time discrete approximation from the exact solution of the continuous diffusion problem, we use u_e for the latter. With u^{k-1} we mean, from now on, the finite element approximation of the solution at time t_{k-1}.

Note that the forms a_0 and L_0 are identical to the forms met in Chapter 2.1.6, except that the unknown now is a scalar field and not a vector field. Instead of solving (2.67) by a finite element method, i.e., projecting I onto V_h via the problem $a_0(u,v) = L_0(v)$, we could simply interpolate u^0 from I. That is, if $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, we simply set $U_j = I(x_j, y_j)$, where (x_j, y_j) are the coordinates of node no. j. We refer to these two strategies as computing the initial condition by either projecting I or interpolating I. Both operations are easy to compute through one statement, using either the `project` or `interpolate` function.

2.3.2 Implementation

Our program needs to perform the time stepping explicitly, but can rely on FEniCS to easily compute a_0, L_0, a, and L, and solve the linear systems for the unknowns. We realize that a does not depend on time, which means that its associated matrix also will be time independent. Therefore, it is wise to explicitly create matrices and vectors as in Chapter 2.1.11. The matrix A arising from a

can be computed prior to the time stepping, so that we only need to compute the right-hand side b, corresponding to L, in each pass in the time loop. Let us express the solution procedure in algorithmic form, writing u for u^k and u_{prev} for the previous solution u^{k-1}:

```
define Dirichlet boundary condition ($u_0$, Dirichlet boundary, etc.)
if $u_{\mathrm{prev}}$ is to be computed by projecting $I$:
  define $a_0$ and $L_0$
  assemble matrix $M$ from $a_0$ and vector $b$ from $L_0$
  solve $MU = b$ and store $U$ in $u_{\mathrm{prev}}$
else: (interpolation)
  let $u_{\mathrm{prev}}$ interpolate $I$
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 $u$
  $t \leftarrow t + \Delta t$
  $u_{\mathrm{prev}} \leftarrow u$ (be ready for next step)
```

Before starting the coding, we shall construct a problem where it is easy to determine if the calculations are correct. The simple backward time difference is exact for linear functions, so we decide to have a linear variation in time. Combining a second-order polynomial in space with a linear term in time,

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{2.73}$$

yields a function whose computed values at the nodes may be exact, regardless of the size of the elements and Δt, as long as the mesh is uniformly partitioned. Inserting (2.73) in the PDE problem (2.61), it follows that u_0 must be given as (2.73) and that $f(x, y, t) = \beta - 2 - 2\alpha$ and $I(x, y) = 1 + x^2 + \alpha y^2$.

A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condition u_0 given by (2.73). Given a `mesh` and an associated function space `V`, we can specify the u_0 function as

```
alpha = 3; beta = 1.2
u0 = Function(V, '1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
              {'alpha': alpha, 'beta': beta})
u0.t = 0
```

This function expression has the components of `x` as independent variables, while `alpha`, `beta`, and `t` are parameters. The parameters can either be set through a dictionary at construction time, as demonstrated for `alpha` and `beta`, or anytime through attributes in the function instance, as shown for the `t` parameter.

The essential boundary conditions, along the whole boundary in this case, are set in the usual way,

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

boundary = Boundary()
bc = DirichletBC(V, u_exact, boundary)
```

The initial condition can be computed by either projecting or interpolating I. The $I(x, y)$ function is available in the program through `u0`, as long as `u0.t` is zero. We can then do

```
u_prev = interpolate(u0, V)
# or
u_prev = project(u0, V)
```

Note that we could, as an equivalent alternative to using `project`, define a_0 and L_0 as we did in Chapter 2.1.6 and form a `VariationalProblem` instance. To actually recover (2.73) to machine precision, it is important not to compute the discrete initial condition by projecting I, but by interpolating I so that the nodal values are exact at $t = 0$ (projection will imply approximative values at the nodes).

The definition of a and L goes as follows:

```
dt = 0.3      # time step

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

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

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

Finally, we perform the time stepping in a loop:

```
u = Function(V)   # the unknown at a new time level
T = 2             # total simulation time
t = dt

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

    t += dt
    u_prev.assign(u)
```

Observe that `u0.t` must be updated before `bc` applies it to enforce the Dirichlet conditions at the current time level.

The time loop above does not contain any examination of the numerical solution, which we must include in order to verify the implementation. As in many previous examples, we compute the difference between the array of nodal values of `u` and the array of the interpolated exact solution. The following code is to be included inside the loop, after `u` is found:

```
u_e = interpolate(u0, V)
maxdiff = (u_e.vector().array() - u.vector().array()).max()
print 'Max error, t=%-10.3f:' % maxdiff
```

The right-hand side vector `b` must obviously be recomputed at each time level. With the construction `b = assemble(L)`, 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 complete program code for this time-dependent case is stored in the file `diffusion2D_D1.py`.

2.3.3 Avoiding 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 $\mathcal{O}(N^{1+p})$, where $p \geq 0$. As $N \to \infty$, the solve operation will for $p > 0$ dominate, 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.

To see how repeated assembly can be avoided, we look at the $L(v)$ form in (2.72), 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 Chapter 2.1.11, 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

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

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

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

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} and

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

and

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

We have immediate access to U^{k-1} in the program since that is the vector in the u_prev 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

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

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 (2.71) 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,

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

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

$$K_{ij} = \int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x\,. \tag{2.74}$$

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

$$a_K(u, v) = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x, \tag{2.75}$$

$$a_M(u, v) = \int_\Omega vu \, \mathrm{d}x, \, . \tag{2.76}$$

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 $b = MU^{k-1} + \Delta t M F^k$ at each time level.

The following modifications are needed in the `diffusion2D_D1.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 $A = M + \Delta K$
4. Define f as a `Function`
5. Interpolate the formula for f to a finite element function F^k
6. Compute $b = MU^{k-1} + \Delta t M F^k$

The relevant code segments become

```
# 1.
a_K = dot(grad(v), grad(u))*dx
a_M = u*v*dx

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

# 4.
f = Function(V, 'beta - 2 - 2*alpha',
             {'beta': beta, 'alpha': alpha})

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

The complete program appears in the file `diffusion2D_D2.py`..

2.3.4 A Physical Example

With the basic programming techniques for time-dependent problem from Chapters 2.3.3–2.3.2 we are ready to attack more physically realistic examples. 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 some box-shaped domain Ω in d dimensions with coordinates $x_0, \ldots, x_{d-1}$ (the problem is meaningful in 1D, 2D, and 3D). At the top of the domain, $x_{d-1} = 0$, we have an oscillating temperature

$$T_0(t) = T_R + T_A \sin(\omega t),$$

where T_R is some reference temperature, T_A is the amplitude of the temperature variations at the surface, and ω is the frequency of the temperature oscillations. At all other boundaries we assume that the temperature does not change anymore when we move away from the boundary, i.e., the normal derivative is zero. Initially, the temperature can be taken as T_R everywhere. The heat conductivity properties of the soil in the ground may vary with space so we introduce a variable coefficient κ reflecting this property. Figure 2.4 shows a sketch of the problem, with a small region where the heat conductivity is much lower.

The initial-boundary value problem for this problem reads

$$\begin{aligned}
\varrho c \frac{\partial T}{\partial t} &= \nabla \cdot (k \nabla T) \text{ in } \Omega \times (0, T], & (2.77)\\
T &= T_0(t) \text{ on } \Gamma_0, & (2.78)\\
\frac{\partial T}{\partial n} &= 0 \text{ on } \partial\Omega \cap \Gamma_0, & (2.79)\\
T &= T_R \text{ at } t = 0\,. & (2.80)
\end{aligned}$$

Here, ϱ is the density of the soil, c is the heat capacity, κ is the thermal conductivity (heat conduction coefficient) in the soil, and Γ_0 is the surface boundary $x_{d-1} = 0$.

We use a θ-scheme in time, i.e., the evolution equation $\partial P/\partial t = Q(t)$ is discretized as

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

where $\theta \in [0, 1]$ is a weighting factor: $\theta = 1$ corresponds to the backward difference scheme, $\theta = 1/2$ to the Crank-Nicolson scheme, and $\theta = 0$ to a forward difference scheme. The θ-scheme applied to our PDE results in

$$\varrho c \frac{T^k - T^{k-1}}{\Delta t} = \theta \nabla \cdot \left(k \nabla T^k\right) + (1-\theta) \nabla \cdot \left(k \nabla T^{k-1}\right)\,.$$

Bringing this time-discrete PDE on weak form follows the technique shown many times earlier in this tutorial. In the standard notation $a(T, v) = L(v)$ the weak

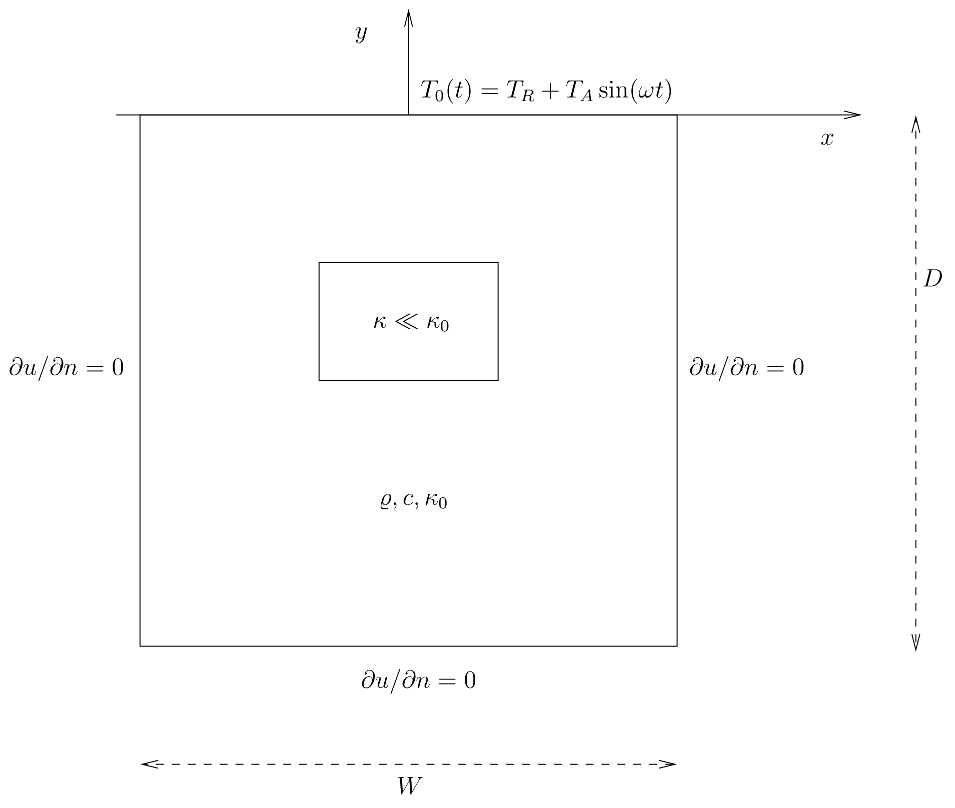

y
$T_0(t) = T_R + T_A \sin(\omega t)$
x
D
$\kappa \ll \kappa_0$
$\partial u/\partial n = 0$
$\partial u/\partial n = 0$
ϱ, c, κ_0
$\partial u/\partial n = 0$
W

Figure 2.4: Sketch of a (2D) problem involving heating and cooling of the ground due to an oscillating surface temperature.

form has

$$
\begin{aligned}
a(T,v) &= \int_\Omega (\varrho c T v + \theta \Delta t \kappa \nabla v \cdot \nabla T)\, \mathrm{d}x, &(2.81)\\
L(v) &= \int_\Omega \left(\varrho c v T^{k-1} - (1-\theta)\Delta t \kappa \nabla v \cdot \nabla T\right)\, \mathrm{d}x\,. &(2.82)
\end{aligned}
$$

Observe that boundary integrals vanish because of the Neumann boundary conditions.

The size of a 3D box is taken as $W \times W \times D$, where D is the depth and $W = D/2$ is the width. We give the degree of the basis functions at the command line, then D, and then the divisions of the domain in the various directions. To make a box, rectangle, or interval of arbitrary (not unit) size, we have the DOLFIN classes Box, Rectangle, and Interval at our disposal. The mesh and the function space can be created by the following code:

```
degree = int(sys.argv[1])
D = float(sys.argv[2])
divisions = [int(arg) for arg in sys.argv[3:]]
d = len(divisions)  # no of space dimensions
if d == 1:
    mesh = Interval(divisions[0], -D, 0)
elif d == 2:
    mesh = Rectangle(0, -D, D/2, 0, divisions[0], divisions[1])
elif d == 3:
    mesh = Box(0, 0, -D, D/2, D/2, 0,
               divisions[0], divisions[1], divisions[2])
V = FunctionSpace(mesh, 'CG', degree)
```

The Rectangle and Box instances are defined by the coordinates of the "minimum" and "maximum" corners.

Setting Dirichlet conditions at the upper boundary can be done by

```
T_R = 0; T_A = 1.0; omega = 2*pi
T_0 = Function(V, 'T_R + T_A*sin(omega*t)',
               {'T_R': T_R, 'T_A': T_A, 'omega': omega, 't': 0.0})

class Surface(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[d-1]) < 1E-14

surface = Surface()
bc = DirichletBC(V, T_0, surface)
```

Quite simple values (non-physical for soil and real temperature variations) are chosen for the initial testing.

The κ function can be defined as a constant κ_1 inside the particular rectangular area with a special soil composition, as indicated in Figure 2.4. Outside this area κ is a constant κ_0. The domain of the rectangular area are taken as

$$[-W/4, W/4] \times [-W/4, W/4] \times [-D/2, -D/2 + D/4]$$

in 3D, with $[-W/4, W/4] \times [-D/2, -D/2 + D/4]$ in 2D and $[-D/2, -D/2 + D/4]$ in 1D. Since we need some testing in the definition of the $\kappa(\boldsymbol{x})$ function, the most straightforward approach is to define a subclass of `Function` with an `eval` method for computing the values:

```
class Kappa(Function):
    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
        d = len(x)  # no of space dimensions
        material = 0  # 0: outside, 1: inside
        if d == 1:
            if -D/2. < x[d-1] < -D/2. + D/4.:
                material = 1
        elif d == 2:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4.:
                material = 1
        elif d == 3:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4. and -W/4. < x[1] < W/4.:
                material = 1
        value[0] = kappa_0 if material == 0 else kappa_1
```

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 nodes in the mesh. 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 involved and not covered here.) Using inline if-tests in C++, we can make string expressions for κ:

```
kappa_0 = 0.2
kappa_1 = 0.001
kappa_str = {}
kappa_str[1] = 'x[0] > -%s/2 && x[0] < -%s/2 + %s/4 ? %g : %g' % \
               (D, D, D, kappa_1, kappa_0)
kappa_str[2] = 'x[0] > -%s/4 && x[0] < %s/4 '\
            '&& x[1] > -%s/2 && x[1] < -%s/2 + %s/4 ? %g : %g' % \
               (W, W, D, D, D, kappa_1, kappa_0)
kappa_str[3] = 'x[0] > -%s/4 && x[0] < %s/4 '\
               'x[1] > -%s/4 && x[1] < %s/4 '\
            '&& x[2] > -%s/2 && x[2] < -%s/2 + %s/4 ? %g : %g' % \
               (W, W, W, W, D, D, D, kappa_1, kappa_0)

kappa = Function(V, kappa_str[d])
```

For example, in 2D `kappa_str[1]` becomes

```
x[0] > -0.5/4 && x[0] < 0.5/4 && x[1] > -1.0/2 &&
x[1] < -1.0/2 + 1.0/4 ? 1e-07 : 0.2
```

for $D = 1$ and $W = D/2$ (the string is one line, but broken into two here to fit the page width). It is very important to have a `D` that is `float` and not `int`, otherwise one gets integer divisions in the C++ expression and a completely wrong κ function.

We are now ready to define the initial condition and the a and L forms of our problem:

```
T_prev = interpolate(Constant(mesh, T_R), V)

rho = 1
c = 1
period = 2*pi/omega
T_stop = 5*period
dt = period/20  # 20 time steps per period
theta = 1

v = TestFunction(V)
T = TrialFunction(V)
f = Constant(mesh, 0)
a = rho*c*T*v*dx + theta*dt*kappa*dot(grad(v), grad(T))*dx
L = (rho*c*T_prev*v + dt*f*v -
     (1-theta)*dt*kappa*dot(grad(v), grad(T)))*dx

A = assemble(a)
b = None  # variable used for memory savings in assemble calls
```

We could, alternatively, break a and L up in subexpressions and assemble a mass matrix and stiffness matrix, as exemplified in Chapter 2.3.3, to avoid assembly of b at every time level. This modification is straightforward and left as an exercise. The speed-up can be significant in 3D problems.

The time loop is very similar to what we have displayed in Chapter 2.3.2:

```
T = Function(V)   # unknown at the current time level
t = dt
while t <= T_stop:
    b = assemble(L, tensor=b)
    T_0.t = t
    bc.apply(A, b)
    solve(A, T.vector(), b)
    # visualization statements
    t += dt
    T_prev.assign(T)
```

The complete code in diffusion123D_sin.py contains several statements related to visualization of the solution, both as a finite element field (plot calls) and as a curve in the vertical direction. The code also plots the exact analytical solution,

$$T(x,t) = T_R + T_A e^{ax}\sin(\omega t + ax), \quad a = \sqrt{\frac{\omega\varrho c}{2\kappa}},$$

which is valid when κ is constant throughout Ω. The reader is encouraged to play around with the code and test out various parameter sets:

- $T_R = 0$, $T_A = 1$, $\kappa_0 = \kappa_1 = 0.2$, $\varrho = c = 1$, $\omega = 2\pi$
- $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.01$, $\varrho = c = 1$, $\omega = 2\pi$
- $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.001$, $\varrho = c = 1$, $\omega = 2\pi$

- $T_R = 10$ C, $T_A = 10$ C, $\kappa_0 = 1.1\ \mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\kappa_0 = 2.3\ \mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\varrho = 1500$ kg/m^3, $c = 1600\ \mathrm{Nm\,kg}^{-1}\mathrm{K}^{-1}$, $\omega = 2\pi/24$ 1/h $= 7.27 \cdot 10^{-5}$ 1/s, $D = 1.5$ m

The latter set of data is relevant for real temperature variations in the ground.

2.4 Controlling the Solution of Linear Systems

Several linear algebra packages, referred to as linear algebra *backends*, can be used in FEniCS to solve linear systems: PETSc, uBLAS, Epetra (Trilinos), or MTL4. Which backend to apply can be controlled by setting

```
parameters['linear algebra backend'] = backendname
```

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

The 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. The underlying PETSc objects 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.

2.4.1 Variational Problem Objects

Let us explain how one can choose between direct and iterative solvers. We have seen that there are two ways of solving linear systems, either we call the `solve()` method in a `VariationalProblem` instance or we call the `solve(A, U, b)` function with the assembled coefficient matrix `A`, right-hand side vector `b`, and solution vector `U`.

In case we use a `VariationalProblem` instance, named `problem`, it has a `parameters` instance that behaves like a Python dictionary, and we can use this object to choose between a direct or iterative solver:

```
problem.parameters['linear_solver'] = 'direct'
# or
problem.parameters['linear_solver'] = 'iterative'
```

Another parameter `'symmetric'` can be set to `True` if the coefficient matrix is symmetric so that a method exploiting symmetry can be utilized. For example, the default iterative solver is GMRES, but when solving a Poisson equation, the iterative solution process will be more efficient by setting the `'symmetry'` parameter so that a Conjugate Gradient is applied.

Having chosen an interative solver, we can invoke a submenu `'krylov_solver'` in the `parameters` object for setting various parameters for the iterative solver (GMRES or Conjugate Gradients, depending on whether the matrix is symmetric or not):

```
itsolver = problem.parameters['krylov_solver'] # short form
itsolver['absolute_tolerance'] = 1E-10
itsolver['relative_tolerance'] = 1E-6
itsolver['divergence_limit'] = 1000.0
itsolver['gmres_restart'] = 50
itsolver['monitor_convergence'] = True
itsolver['report'] = True
```

Here, `'divergence_limit'` governs the maximum allowable number of iterations, the `'gmres_restart'` parameter tells how many iterations GMRES performs before it restarts, `'monitor_convergence'` prints detailed information about the development of the residual of a solver, `'report'` governs whether a one-line report about the solution method and the number of iterations is written on the screen or not. The absolute and relative tolerances enter (usually residual-based) stopping criteria, which are dependent on the implementation of the underlying iterative solver in the actual backend.

When direct solver is chosen, there is similarly a submenu `'lu_solver'` to set parameters, but here only the `'report'` parameter is available (since direct solvers very soldom have any adjustable parameters). For nonlinear problems there is also submenu `'newton_solver'` where tolerances, maximum iterations, and so on, for a the Newton solver in `VariationalProblem` can be set.

A complete list of all parameters and their default values is printed to the screen by

```
info(problem.parameters, True)
```

2.4.2 Solve Function

For the `solve(A, x, b)` approach, a 4th argument to `solve` determines the type of method:

- `'lu'` for a sparse direct (LU decomposition) method,
- `'cg'` for the Conjugate Gradient (CG) method, which is applicable if `A` is symmetric and positive definite,
- `'gmres'` for the GMRES iterative method, which is applicable when `A` is nonsymmetric,

- `'bicgstab'` for the BiCGStab iterative method, which is applicatble when `A` is nonsymmetric.

The default solver is `'lu'`.

Good performance of an iterative method requires preconditioning of the linear system. The 5th argument to `solve` determines the preconditioner:

- `'none'` for no preconditioning.
- `'jacobi'` for the simple Jacobi (diagonal) preconditioner,
- `'sor'` for SOR preconditioning,
- `'ilu'` for incomplete LU factorization (ILU) preconditioning,
- `'icc'` for incomplete Cholesky factorization preconditioning (requires `A` to be symmetric and positive definite),
- `'amg_hypre'` for algebraic multigrid (AMG) preconditioning with the Hypre package (if available),
- `'mag_ml'` for algebraic multigrid (AMG) preconditioning with the ML package from Trilinos (if available),
- `'default_pc'` for a default preconditioner, which depends on the linear algebra backend (`'ilu'` for PETSc).

If the 5th argument is not provided, `'ilu'` is taken as the default preconditioner.

Here are some sample calls to `solve` demonstrating the choice of solvers and preconditioners:

```
solve(A, u.vector(), b)       # 'lu' is default solver
solve(A, u.vector(), cg)      # CG with ILU prec.
solve(A, u.vector(), 'gmres', 'amg_ml')  # GMRES with ML prec.
```

2.4.3 Setting the Start Vector

The choice of start vector for the iterations in a linear solver is often important. With the `solve(A, U, b)` function the start vector is the vector we feed in for the solution. A start vector with random numbers in the interval $[-1, 1]$ can be computed as

```
n = u.vector().array().size
u.vector()[:] = numpy.random.uniform(-1, 1, n)
solve(A, u.vector(), b, cg, ilu)
```

Or if a `VariationalProblem` instance is used, its `solve` method may take an optional `u` function as argument (which we can fill with the right values):

```
problem = VariationalProblem(a, L, bc)
n = u.vector().array().size
u.vector()[:] = numpy.random.uniform(-1, 1, n)
u = problem.solve(u)
```

The program `Poisson2D_DN_laprm.py` demonstrates the various control mechanisms for steering linear solvers as described above.

2.4.4 Using a Backend-Specific Solver

Here is a demo where we operate on Trilinos-specific vectors, matrices, iterative solvers, and preconditioners. Given a linear system $AU = b$, represented by a `Matrix` instance `A`, and two `Vector` instances `U` and `b`, 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.

```
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 dolfin 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)
```

2.5 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.

2.5.1 Built-In Mesh Generation Tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitInterval`, `UnitSphere`, `UnitSquare`, `Interval`, `Rectangle`, `Box`, `UnitCircle`, and `UnitCube`. 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 = UnitInterval(20)     # 20 cells, 21 vertices
mesh = Interval(20, -1, 1)  # domain [-1,1]

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

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

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

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

# 10 divisions in radial directions
mesh = UnitCircle(10)
mesh = UnitSphere(10)
```

2.5.2 Transforming Mesh Coordinates

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,

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^s \tag{2.83}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{2.84}$$

toward $x = b$

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 = Rectangle(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):
```

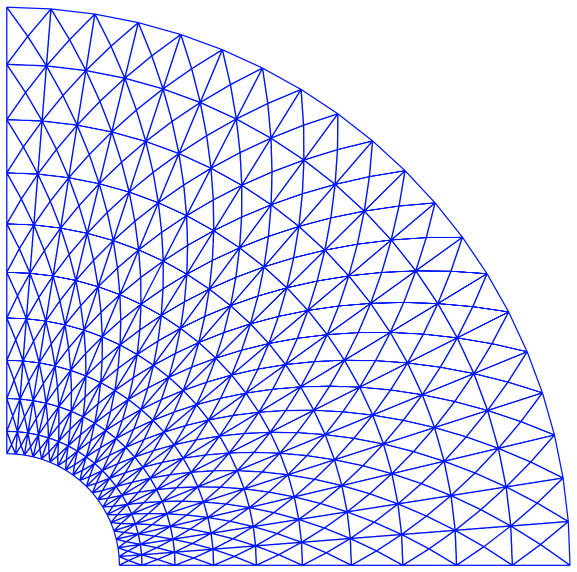


Figure 2.5: Hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

```
        return [r*cos(Theta*s), r*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 convention an $M \times 2$ array so we need to take the transpose. The resulting mesh is displayed in Figure 2.5.

2.5.3 Separate Preprocessor Applications

2.6 Handling Domains with Different Materials

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, this 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

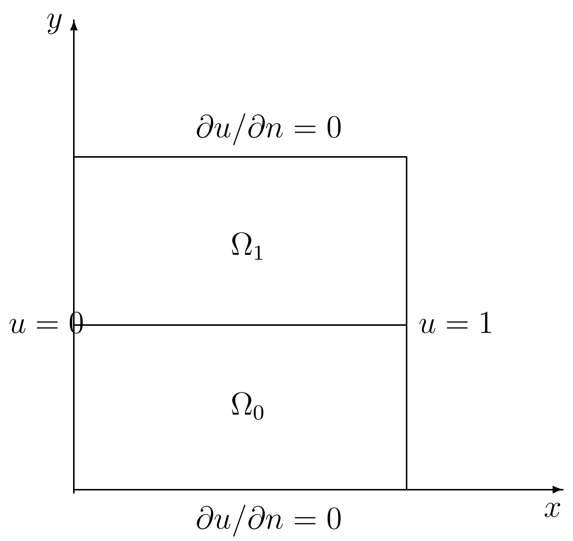


Figure 2.6: Sketch of a Poisson problem with a variable coefficient that is constant in each of the two subdomains Ω_0 and Ω_1.

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. Later, a multi-material problem in d space dimensions is addressed.

2.6.1 Working with Two Subdomains

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{2.85}$$

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 2.6,

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

We define $k(x,y) = 1$ in Ω_0 and $k(x,y) = 10$ in Ω_1. As boundary conditions, we choose $u = 0$ at $x = 0$, $u = 1$ at $x = 1$, and $\partial u/\partial n = 0$ at $y = 0$ and $y = 1$. This choice implies the simple solution $u(x,y) = x$, which we should recover exactly with linear or higher order finite elements.

Physically, the present problem may correspond to heat conduction, where the heat conduction in Ω_1 is ten times more efficient than in Ω_0. An alternative interpretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differs by a factor of 10.

2.6.2 The Implementation

The new functionality in this subsection regards how to to define the subdomains Ω_0 and Ω_1. Defining a subdomain is done by creating a subclass of `SubDomain` and implementing the `inside` function. In the present case we define

```
class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] <= 0.5 else False

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] >= 0.5 else False
```

The next task is to introduce a `MeshFunction` 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` is a discrete function that can be evaluated at a set of so-called *mesh entities*. Three 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.

Since we need to define subdomains of Ω in the present example, we must make use of a `MeshFunction` over cells. The `MeshFunction` constructor is fed with three 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` instance, and 3) the topological dimension of the mesh entity in question: cells have topological dimension equal to the number of space dimensions in the PDE problem, and facets have one dimension lower. Alternatively, the constructor can take just a filename and initialize the `MeshFunction` from data in a file. We shall demonstrate this functionality in the next multi-material problem in Chapter 2.7.

We start with creating a `MeshFunction` whose values are non-negative integers (`'uint'`) for numbering the subdomains. The mesh entities of interest are the cells, which have dimension 2 in a two-dimensional problem (1 in 1D, 3 in 3D). The appropriate code for defining the `MeshFunction` for two subdomains then reads

```
subdomains = MeshFunction('uint', mesh, 2)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(subdomains, 0)
subdomain1 = Omega1()
subdomain1.mark(subdomains, 1)
```

Calling `subdomains.values()` returns a `numpy` array of the subdomain values. That is, `subdomain.values()[i]` is the subdomain value of cell no. `i`.

This array is used to look up the subdomain or material number of a specific element.

Now we want a function k that is piecewise 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 are 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 subdomain.values()), 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(subdomains.values())):
    subdomain_no = subdomains.values()[cell_no]
    k.vector()[cell_no] = k_values[subdomain_no]
```

Long loops in Python are known to be slow, so for large meshes the 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 subdomain.values(), but where the value i of an entry in subdomain.values() is replaced by k_values[i]. Such an operation is carried out by the numpy function choose:

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

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

Having the k function ready for finite element computations, we can proceed in the normal manner with defining essential boundary conditions, as in Chapter 2.1.10, and the $a(u, v)$ and $L(v)$ forms, as in Chapter 2.1.12. All the details can be found in the file Poisson2D_2mat.py.

2.6.3 Multiple Neumann, Robin, and Dirichlet Conditions

Let us go back to the model problem from Chapter 2.1.10 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 cell 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.

Essentially, we still stick to the model problem from Chapter 2.1.10, but replace the Neumann condition at $y = 0$ by a *Robin condition*[6]:

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

where p and q are specified functions. Since we have prescribed a simple solution in our model problem, $u = 1 + x^2 + y^4$, we adjust p and q such that the condition holds at $y = 0$. This implies that $q = 1 + x^2 + 2y^2$ and p can be arbitrary (the normal derivative at $y = 0$: $\partial u/\partial n = -\partial u/\partial y = -4y = 0$).

Now we have four parts of the boundary: Γ_N which corresponds to the upper side $y = 1$, Γ_R which corresponds to the lower part $y = 0$, Γ_0 which corresponds to the left part $x = 0$, and Γ_1 which corresponds to the right part $x = 1$. The complete boundary-value problem reads

$$\begin{aligned} -\Delta u &= -6 \text{ in } \Omega, & (2.86)\\ u &= u_L \text{ on } \Gamma_0, & (2.87)\\ u &= u_R \text{ on } \Gamma_1, & (2.88)\\ -\frac{\partial u}{\partial n} &= p(u-q) \text{ on } \Gamma_R, & (2.89)\\ -\frac{\partial u}{\partial n} &= 4y \text{ on } \Gamma_N\,. & (2.90) \end{aligned}$$

The involved prescribed functions are $u_L == 1+2y^2$, $u_R = 2+2y^2$, $q = 1+x^2+2y^2$, p is arbitrary, and $g = -4y$.

Integration by parts of $-\int_\Omega v$
$Deltau\,\mathrm{d}x$ becomes as usual

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

The boundary integral vanishes on $\Gamma_0\cup\Gamma_1$, and we split the parts over Γ_N and Γ_R since we have different conditions at those parts:

$$-\int_{\partial\Omega} v\frac{\partial u}{\partial n}\,\mathrm{d}s = -\int_{\Gamma_N} v\frac{\partial u}{\partial n}\,\mathrm{d}s - \int_{\Gamma_R} v\frac{\partial u}{\partial n}\,\mathrm{d}s = \int_{\Gamma_N} vg\,\mathrm{d}s + \int_{\Gamma_R} vp(u-q)\,\mathrm{d}s\,.$$

[6]The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law.

The weak form then becomes

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x + \int_{\Gamma_N} gv \,\mathrm{d}s + \int_{\Gamma_R} vp(u-q) \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x,$$

We want to write this weak form in the standard notation $a(u, v) = L(v)$, which requires that we indentify 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:

$$\int_{\Gamma_R} vp(u-q) \,\mathrm{d}s = \int_{\Gamma_R} vpu \,\mathrm{d}s - \int_{\Gamma_R} vpq \,\mathrm{d}s\,.$$

We then have

$$a(u, v) = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x + \int_{\Gamma_R} vpu \,\mathrm{d}s, \tag{2.91}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s + \int_{\Gamma_R} vpq \,\mathrm{d}s\,. \tag{2.92}$$

A natural starting point for implementation is the `Poisson2D_DN2.py` program, which we now copy to `Poisson2D_DNR.py`. The new aspects are

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.

Task 1 makes use of the `MeshFunction` object, but contrary to Chapter 2.6.2, this is not a function over cells, but a function over cell facets. The topological dimension of cell facets is one lower than the cell interiors, so in a two-dimensional problem the dimension becomes 1. In general, the facet dimension is given as `mesh.topology().dim()-1`, which we use in the code for ease of direct reuse in other problems. The construction of a `MeshFunction` instance to mark boundary parts now reads

```
boundary_parts = \
  MeshFunction("uint", mesh, mesh.topology().dim()-1)
```

As in Chapter 2.6.2 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $y = 0$ boundary can be marked by

```
class LowerRobinBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[1]) < tol

Gamma_R = LowerRobinBoundary()
Gamma_R.mark(boundary_parts, 0)
```

The code for the $y = 1$ boundary is similar and is seen in `Poisson2D_DNR.py`.

The Dirichlet boundaries are marked similarly, using subdomain number 2 for Γ_0 and 3 for Γ_1:

```
class LeftDirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0]) < tol

Gamma_0 = LeftDirichletBoundary()
Gamma_0.mark(boundary_parts, 2)

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

Gamma_1 = RightDirichletBoundary()
Gamma_1.mark(boundary_parts, 3)
```

Specifying the `DirichletBC` instances may now make use of the mesh function (instead of a `SubDomain` subclass object) and an indicator for which subdomain each condition should be applied to:

```
u_L = Function(V, '1 + 2*x[1]*x[1]')
u_R = Function(V, '2 + 2*x[1]*x[1]')
bc = [DirichletBC(V, u_L, boundary_parts, 2),
      DirichletBC(V, u_R, boundary_parts, 3)]
```

Some functions need to be defined before we can go on with the `a` and `L` of the variational problem:

```
q = Function(V, '1 + x[0]*x[0] + 2*x[1]*x[1]')
p = Constant(mesh, 100)  # arbitrary function can go here
v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, -6.0)
```

The new aspect of the variational problem is the two distinct boundary integrals. Having a mesh function over exterior cell facets (i.e., 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.

The variational problem can be defined as

```
a = dot(grad(v), grad(u))*dx + v*p*u*ds(0)
L = v*f*dx - v*g*ds(1) + v*p*q*ds(0)
```

For the `ds(0)` and `ds(1)` symbols to work we must obviously connect them (or `a` and `L`) to the mesh function marking parts of the boundary. This is done by a certain keyword argument to the `assemble` function:

```
A = assemble(a, exterior_facet_domains=boundary_parts)
b = assemble(L, exterior_facet_domains=boundary_parts)
```

Then essential boundary conditions are enforced, and the system can be solved in the usual way:

```
for condition in bc: condition.apply(A, b)
u = Function(V)
solve(A, u.vector(), b)
```

At the time of this writing, it is not possible to perform integrals over different parts of the domain or boundary using the `assemble_system` function or the `VariationalProblem` instance.

2.7 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.

2.7.1 The PDE Problem

We generalize the problem in Chapter 2.6.1 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 2.7. 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 Chapter 2.1.14, but the layer boundaries are planes $x_0 =$ const and u varies with x_0 only.

The PDE reads

$$\nabla \cdot (k\nabla u) = 0\,. \tag{2.93}$$

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 2.7 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, while Neumann conditions $\partial u/\partial n = 0$ are set on the remaining boundaries. The complete boundary-value problem is then

$$\begin{array}{rcll} \nabla \cdot (k(x_0)\nabla u(x_0, \ldots, x_{d-1})) &=& 0 & \text{in } \Omega, \\ u &=& 0 & \text{on } \Gamma_0, \\ u &=& 1 & \text{on } \Gamma_1, \\ \frac{\partial u}{\partial n} &=& 0 & \text{on } \Gamma_N\,. \end{array} \tag{2.94}$$

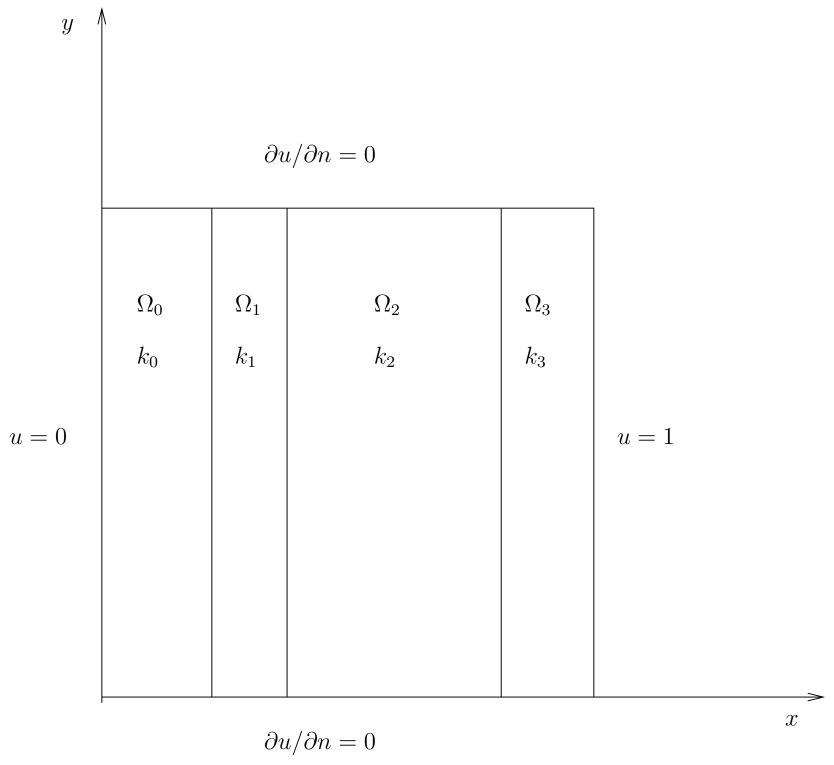


Figure 2.7: Sketch of a multi-material problem.

The domain Ω is divided into s materials Ω_i, $i = 0, \ldots, s-1$, where

$$\Omega_i = \{(x_0, \ldots, x_{d-1} \,|\, L_i \leq x_0 < L_{i+1}\}$$

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 (2.94) is

$$u(x_0, \ldots, x_{d-1}) = \frac{\int_0^{x_0} (k(\tau))^{-1} d\tau}{\int_0^1 (k(\tau))^{-1} d\tau}\,.$$

For a piecewise constant $k(x_0)$ as explained, we get

$$u(x_0, \ldots, x_{d-1}) = \frac{(x_0 - L_i)k_i^{-1} + \sum_{j=0}^{i-1}(L_{j+1} - L_j)k_j^{-1}}{\sum_{j=0}^{s-1}(L_{j+1} - L_j)k_j^{-1}}, \quad L_i \leq x_0 \leq L_{i+1}\,. \tag{2.95}$$

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, any standard finite element method will reproduce this exact solution to machine precision, which is ideal for a test case.

2.7.2 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 Chapter 2.1.14. This results in an instance `mesh` representing the interval $[0, 1]$ in 1D, the unit square in 2D, or the unit cube in 3D.

The subdomains Ω_i must be defined through subclasses of `SubDomain`. Would 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 Chapter 2.6.1 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 elements, and many of these will lie on the subdomain boundaries. All vertices `x` in a cell must be lead to a `True` return value from `inside` for the cell to be a part of a subdomain.

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` instance `subdomains`. The subdomain number of mesh entity number `e`, here cell `e`, is given by `subdomains.values()[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 `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 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_exact(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_exact(xi) for xi in x])
    print u
    plot(x, u)
""" % (L, k))
f.close()
```

2.7.3 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 Chapter 2.6.2, a k that is constant in each element is suitable. We then follow the recipe from Chapter 2.6.2 to compute k:

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

# Vectorized calculation
help = numpy.asarray(subdomains.values(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The essential boundary conditions are defined in the same way is in `Poisson2D_DN2.py` from Chapter 2.1.10 and therefore not repeated here. The variational problem is defined and solved in a standard manner,

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

```
f = Constant(mesh, 0)
a = k*dot(grad(v), grad(u))*dx
L = v*f*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.values())):
    k_meshfunc.values()[i] = k_values[subdomains.values()[i]]

# Vectorized version
help = numpy.asarray(subdomains.values(), dtype=numpy.int32)
k_meshfunc.values()[:] = numpy.choose(help, k_values)

plot(k_meshfunc, title='k as mesh function')
```

The file `Poisson_layers.py` contains the complete code.

2.8 Miscellaneous Topics

2.8.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenics.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 Chapter 2.1.7).

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 = UnitInterval(10)` creates an instance of class `UnitInterval`, which is reached by the the name `mesh`. (Class `UnitInterval` 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.meth`

`self` parameter (Python): required first parameter in class methods, representing a particular instance 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 `X`. 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.att`

2.8.2 Overview of Objects and Functions

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

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquare`, `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.

`UnitSquare(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.

`UnitInterval`, `UnitCube`, `UnitCircle`, `UnitSphere`, `Interval`, `Rectangle`, and `Box`: generate mesh over domains of simple geometric shape, see Chapter 2.5.

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

`Function(V, expression)`: a scalar- or vector-valued function, given as a mathematical formula `expression` (string) written in C++ syntax.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`.

`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 Chapter 2.1.3) for telling whether a point `x` is inside the subdomain or not.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Chapter 2.6.1) or for boundary conditions (see Chapter 2.6.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

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

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

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

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

`VariationalProblem(a, L, bc)`: define a variational problem by a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` instances stored in `bc`.

`VariationalProblem(a, L, bc)`: define and solve a variational problem, given a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` instances stored in `bc`. A 4th argument, `nonlinear=True`, can be given to define and solve nonlinear variational problems (see Chapter 2.2.4).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` instance), `U` as unknown (`Vector` instance), and `b` as right-hand side (`Vector` instance). Usually, `U` is replaced by `u.vector()`, where `u` is a `Function` instance 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 the Viper 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`.

2.8.3 Installing FEniCS

The FEniCS software components are available for Linux, Windows and Mac OS X platforms. Detailed information on how to get FEniCS running on such machines are available at the `fenics.org` website. Here are just some quick descriptions and recommendations by the author.

To make the installation of FEniCS as painless and reliable as possible, the reader is strongly recommended to use Ubuntu Linux. Any standard PC can easily be equipped with Ubuntu Linux, which may live side by side with either Windows or Mac OS X or another Linux installation. Basically, you download Ubuntu from `www.ubuntu.com/getubuntu/download`, burn the file on a CD, reboot the machine with the CD, and answer some usually straightforward questions (if necessary). Ubuntu is quite similar to both Windows 7 and Mac OS X, but to be efficient when doing science with FEniCS this author recommends to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more user friendly.

Instead of making it possible to boot your machine with the Linux Ubuntu operating system, you can run Ubuntu in a separate window in your existing operation system. On Mac, you can use the VirtualBox software available from `http://www.virtualbox.org` to run Ubuntu. On Windows, Wubi makes a tool that automatically installs Ubuntu on the machine. Just give a username and password for the Ubuntu installation, and Wubi performs the rest. You can also use VirtualBox on Windows machines.

Once the Ubuntu window is up and running, go to the `fenics.org` cite and paste in the five few lines that are needed to install what you need.

2.8.4 Books on the Finite Element Method

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering “structural analysis” version of the method. FEniCS builds heavily on concepts in the former version. Readers who prefer mathematical rigor, and analysis of the finite element method, will appreciate the texts by Brenner and Scott (Brenner and Scott, 2008), Braess (Braess, 2007), or Ciarlet (Ciarlet, 2002b). Alternative books with less mathematical rigor and more intuitive and engineering-oriented expositions are ....

2.8.5 Books on Python

Two very popular introductory books on Python are “Learning Python” by Lutz (Lutz, 2007) and “Practical Python” by Hetland (Hetland, 2002). More advanced and comprehensive books include “Programming Python” by Lutz (Lutz, 2006), and “Python Cookbook” (Martelli and Ascher, 2005) and “Python in a Nutshell” (Martelli, 2006) by Martelli. The web page `http:://python.org/...` lists numerous additional books. Very few texts teach Python in a mathematical and numerical context, but the references (Langtangen, 2009a,b) are exceptions.

2.8.6 User-Defined Function Objects

User-defined functions are implemented as a `Function` object. There are several ways of constructing such objects, such as

- a string containing a mathematical expression,
- a subclass of `Function` in Python,
- a subclass of `Function` in C++,
- a set of degrees of freedom of a finite element function.

Writing `pydoc dolfin.Function` gives a documentation of the various techniques.

When defining a function in terms of a mathematical expression inside a string formula, the expression will be turned into a C++ function and compiled to gain efficiency. Therefore, the syntax used in the expression must be valid C++ syntax. Most Python syntax for mathematical expressions are also valid C++ syntax, but power expressions make an exception: `p**a` must be written as `pow(p,a)` in C++ (this is also an alternative Python syntax). The following mathematical functions can be used directly in C++ expressions for `Function` objects: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`, and `fmod`. Moreover, the number π is available as the symbol `pi`. All the listed functions are taken from the `cmath` C++ header file, and one may hence consult documentation of `cmath` for more information on the various functions.

Acknowledgments. The author is very thankful to Johan Hake, Anders Logg, Kent-Andre Mardal, and Kristian Valen-Sendstad for promptly answering all my questions about FEniCS functionality and for implementing all my requests. Professor Douglas Arnold is thanked for valuable feedback on this tutorial.

Part I

Methodology

CHAPTER 3

The Finite Element Method

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-7]**

The finite element method has grown out of Galerkin's method, emerging as a universal method for the solution of differential equations. Much of the success of the finite element method can be contributed to its generality and simplicity, allowing a wide range of differential equations from all areas of science to be analyzed and solved within a common framework. Another contributing factor to the success of the finite element method is the flexibility of formulation, allowing the properties of the discretization to be controlled by the choice of finite element approximating spaces.

In this chapter, we review the finite element method and introduce some basic concepts and notation. In the coming chapters, we discuss these concepts in more detail, with a particular focus on the implementation and automation of the finite element method as part of the FEniCS project.

3.1 A Simple Model Problem

In 1813, Siméon Denis Poisson (Figure 3.1) published in *Bulletin de la société philomatique* his famous equation as a correction of an equation published earlier by Pierre-Simon Laplace. Poisson's equation is a second-order partial differential equation stating that the negative Laplacian $-\Delta u$ of some unknown field $u = u(x)$ is equal to a given function $f = f(x)$ on a domain $\Omega \subset \mathbb{R}^d$, possibly amended by a set of boundary conditions for the solution u on the boundary $\partial\Omega$

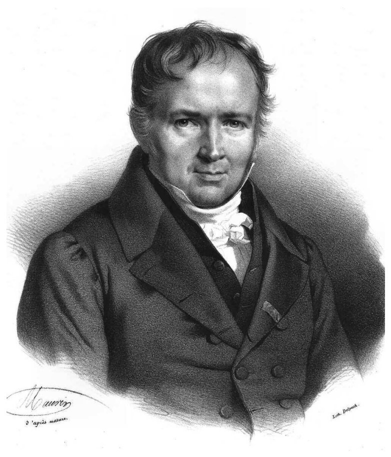


Figure 3.1: Siméon Denis Poisson (1781–1840), inventor of Poisson's equation.

of Ω:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_0 \quad \text{on } \Gamma_D \subset \partial\Omega, \\ -\partial_n u &= g \quad \text{on } \Gamma_N \subset \partial\Omega. \end{aligned} \tag{3.1}$$

The Dirichlet boundary condition $u = u_0$ signifies a prescribed value for the unknown u on a subset Γ_D of the boundary and the Neumann boundary condition $-\partial_n u = g$ signifies a prescribed value for the (negative) normal derivative of u on the remaining boundary $\Gamma_N = \partial\Omega \setminus \Gamma_D$. Poisson's equation is a simple model for gravity, electromagnetism, heat transfer, fluid flow, and many other physical processes. It also appears as the basic building block in a large number of more complex physical models, including the Navier–Stokes equations that we return to below in Chapters **??**.

To derive Poisson's equation (3.1), we may consider a model for the temperature u in a body occupying a domain Ω subject to a heat source f. Letting $\sigma = \sigma(x)$ denote heat flux, it follows by conservation of energy that the outflow of energy over the boundary $\partial\omega$ of any test volume $\omega \subset \Omega$ must be balanced by the energy transferred from the heat source f,

$$\int_{\partial\omega} \sigma \cdot n \,\mathrm{d}s = \int_{\omega} f \,\mathrm{d}x.$$

Integrating by parts, it follows that

$$\int_{\omega} \nabla \cdot \sigma \,\mathrm{d}x = \int_{\omega} f \,\mathrm{d}x$$

for all test volumes ω and thus that $\nabla \cdot \sigma = f$ (by suitable regularity assumptions on σ and f). If we now make the assumption that the heat flux σ is proportional

missing figure to be added

Figure 3.2: Poisson's equation is a simple consequence of balance of energy in an arbitrary test volume $\omega \subset \Omega$.

to the negative gradient of the temperature u (Fourier's law),

$$\sigma = -\kappa \nabla u,$$

we arrive at the following system of equations:

$$\begin{aligned} \nabla \cdot \sigma &= f \quad \text{in } \Omega, \\ \sigma + \nabla u &= 0 \quad \text{in } \Omega, \end{aligned} \tag{3.2}$$

where we have assumed that the the heat conductivity $\kappa = 1$. Replacing σ in the first of these equations by $-\nabla u$, we arrive at Poisson's equation (3.1). We note that one may as well arrive at the system of first order equations (3.2) by introducing $\sigma = -\nabla u$ as an auxiliary variable in the second order equation (3.1). We also note that the Dirichlet and Neumann boundary conditions in (3.1) correspond to prescribed values for the temperature and heat flux respectively.

3.2 Finite Element Discretization

3.2.1 Discretizing Poisson's equation

To discretize Poisson's equation (3.1) by the finite element method, we first multiply by a test function v and integrate by parts to obtain

$$\int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x - \int_\Omega v \, \partial_n u \, \mathrm{d}s = \int_\Omega v f \, \mathrm{d}x.$$

Letting the test function v vanish on the Dirichlet boundary Γ_D where the solution u is known, we arrive at the following classical variational problem: Find $u \in V$ such that

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x = \int_\Omega v f \,\mathrm{d}x - \int_{\Gamma_N} v\, g \,\mathrm{d}s \quad \forall v \in \hat{V}. \tag{3.3}$$

The test space $\hat{V}$ is defined by

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\},$$

and the trial space V contains members of $\hat{V}$ shifted by the Dirichlet condition,

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_D\}.$$

We may now discretize Poisson's equation by restricting the variational problem (3.3) to a pair of discrete spaces: Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla v \cdot \nabla u_h \,\mathrm{d}x = \int_\Omega v f \,\mathrm{d}x - \int_{\Gamma_N} v\, g \,\mathrm{d}s \quad \forall v \in \hat{V}_h \subset \hat{V}. \tag{3.4}$$

We note here that the Dirichlet condition $u = u_0$ on Γ_D enters into the definition of the trial space V_h (it is an *essential* boundary condition), whereas the Neumann condition $-\partial_n u = g$ on Γ_N enters into the variational problem (it is a *natural* boundary condition).

To solve the discrete variational problem (3.4), we must construct a suitable pair of discrete test and trial spaces $\hat{V}_h$ and V_h. We return to this issue below, but assume for now that we have a basis $\{\hat{\phi}_i\}_{i=1}^N$ for $\hat{V}_h$ and a basis $\{\phi_j\}_{j=1}^N$ for V_h. We may then make an ansatz for u_h in terms of the basis functions of the trial space,

$$u_h = \sum_{j=1}^N U_j \phi_j,$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom to be computed. Inserting this into (3.4) and varying the test function v over the basis functions of the discrete test space $\hat{V}_h$, we obtain

$$\sum_{j=1}^N U_j \int_\Omega \nabla \hat{\phi}_i \cdot \nabla \phi_j \,\mathrm{d}x = \int_\Omega \hat{\phi}_i f \,\mathrm{d}x - \int_{\Gamma_N} \hat{\phi}_i g \,\mathrm{d}s, \quad i = 1, 2, \ldots, N.$$

We may thus compute the finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ by solving the linear system

$$AU = b,$$

where

$$A_{ij} = \int_\Omega \nabla \hat{\phi}_i \cdot \nabla \phi_j \,\mathrm{d}x,$$
$$b_i = \int_\Omega \hat{\phi}_i f \,\mathrm{d}x - \int_{\Gamma_N} \hat{\phi}_i g \,\mathrm{d}s.$$

3.2.2 Discretizing the first order system

We may similarly discretize the first order system (3.2) by multiplying the first equation by a test function v and the second by a test function τ. Summing up and integrating by parts, we find

$$\int_\Omega v\nabla\cdot\sigma + \tau\cdot\sigma - \nabla\cdot\tau\, u\,\mathrm{d}x + \int_{\partial\Omega} \tau\cdot n\, u\,\mathrm{d}s = \int_\Omega vf\,\mathrm{d}x \quad \forall v\in\hat{V}.$$

The normal flux $\sigma\cdot n = g$ is known on the Neumann boundary Γ_N so we may take $\tau\cdot n = 0$ on Γ_N. Inserting the value for u on the Dirichlet boundary Γ_D, we thus arrive at the following variational problem: Find $(u,\sigma)\in V$ such that

$$\int_\Omega v\nabla\cdot\sigma + \tau\cdot\sigma - \nabla\cdot\tau\, u\,\mathrm{d}x = \int_\Omega vf\,\mathrm{d}x - \int_{\Gamma_D} \tau\cdot n\, u_0\,\mathrm{d}s \quad \forall (v,\tau)\in\hat{V}. \tag{3.5}$$

Now, $\hat{V}$ and V are a pair of suitable test and trial spaces, here

$$\begin{aligned}\hat{V} &= \{(v,\tau) : v\in L^2(\Omega), \tau\in H(\mathrm{div};\Omega), \tau\cdot n = 0 \text{ on } \Gamma_N\},\\ V &= \{(v,\tau) : v\in L^2(\Omega), \tau\in H(\mathrm{div};\Omega), \tau\cdot n = g \text{ on } \Gamma_N\}.\end{aligned}$$

As above, we restrict this variational problem to a pair of discrete test and trial spaces $\hat{V}_h\subset\hat{V}$ and $V_h\subset V$ and make an ansatz for the finite element solution of the form

$$(u_h,\sigma_h) = \sum_{j=1}^N U_j(\phi_j,\psi_j),$$

where $\{(\phi_j,\psi_j)\}_{j=1}^N$ is a basis for the trial space V_h. Typically, either ϕ_j or ψ_j will vanish, so that the basis is really the tensor product of a basis for an L^2 space with an $H(\mathrm{div})$ space. We thus obtain a linear system for the degrees of freedom $U\in\mathbb{R}^N$ by solving a linear system $AU=b$, where now

$$\begin{aligned}A_{ij} &= \int_\Omega \hat{\phi}_i\nabla\cdot\psi_j + \hat{\psi}_i\cdot\psi_j - \nabla\cdot\hat{\psi}_i\,\phi_j\,\mathrm{d}x,\\ b_i &= \int_\Omega \hat{\phi}_i f\,\mathrm{d}x - \int_{\Gamma_D}\hat{\psi}_i\cdot n\, u_0\,\mathrm{d}s.\end{aligned}$$

We note that the variational problem (3.5) differs from the variational problem (3.3) in that the Dirichlet condition $u = u_0$ on Γ_D enters into the variational formulation (it is now a natural boundary condition), whereas the Neumann condition $\sigma = g$ on Γ_N enters into the definition of the trial space V (it is now an essential boundary condition).

Such mixed methods require some care in selecting spaces that discretize L^2 and $H(\mathrm{div})$ in a compatible way. Stable discretizations must satisfy the so-called *inf–sup* or Ladysenskaja–Babuška–Brezzi (LBB) condition. This theory explains why many of the elements for mixed methods seem complicated compared to those for standard Galerkin methods.

3.3 Finite Element Abstract Formalism

3.3.1 Linear problems

We saw above that the finite element solution of Poisson's equation (3.1) or (3.2) can be obtained by restricting an infinite dimensional variational problem to a finite dimensional variational problem and solving a linear system.

To formalize this, we consider a general linear variational problem written in the following canonical form: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \tag{3.6}$$

where $\hat{V}$ is the test space and V is the trial space. We may thus express the variational problem in terms of a *bilinear form* a and *linear form* (functional) L,

$$\begin{aligned} a &: \hat{V} \times V \to \mathbb{R}, \\ L &: \hat{V} \to \mathbb{R}. \end{aligned}$$

As above, we discretize the variational problem (3.6) by restricting to a pair of discrete test and trial spaces: Find $u_h \in V_h \subset V$ such that

$$a(v, u_h) = L(v) \quad \forall v \in \hat{V}_h \subset \hat{V}. \tag{3.7}$$

To solve the discrete variational problem (3.7), we make an ansatz of the form

$$u_h = \sum_{j=1}^{N} U_j \phi_j, \tag{3.8}$$

and take $v = \hat{\phi}_i$, $i = 1, 2, \ldots, N$, where $\{\hat{\phi}_i\}_{i=1}^N$ is a basis for the discrete test space $\hat{V}_h$ and $\{\phi_j\}_{j=1}^N$ is a basis for the discrete trial space V_h. It follows that

$$\sum_{j=1}^{N} U_j \, a(\hat{\phi}_i, \phi_j) = L(\hat{\phi}_i), \quad i = 1, 2, \ldots, N.$$

We thus obtain the degrees of freedom U of the finite element solution u_h by solving a linear system $AU = b$, where

$$\begin{aligned} A_{ij} &= a(\hat{\phi}_i, \phi_j), \quad i, j = 1, 2, \ldots, N, \\ b_i &= L(\hat{\phi}_i). \end{aligned} \tag{3.9}$$

3.3.2 Nonlinear problems

We also consider nonlinear variational problems written in the following canonical form: Find $u \in V$ such that

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \tag{3.10}$$

where now $F : V \times \hat{V} \to \mathbb{R}$ is a *semilinear* form, linear in the argument(s) subsequent to the semicolon. As above, we discretize the variational problem (3.10) by restricting to a pair of discrete test and trial spaces: Find $u_h \in V_h \subset V$ such that

$$F(u_h; v) = 0 \quad \forall v \in \hat{V}_h \subset \hat{V}.$$

The finite element solution $u_h = \sum_{j=1}^{N} U_j \phi_j$ may then be computed by solving a nonlinear system of equations,

$$b(U) = 0, \tag{3.11}$$

where $b : \mathbb{R}^N \to \mathbb{R}^N$ and

$$b_i(U) = F(u_h; \hat{\phi}_i), \quad i = 1, 2, \ldots, N. \tag{3.12}$$

To solve the nonlinear system (3.11) by Newton's method or some variant of Newton's method, we compute the Jacobian $A = b'$. We note that if the semilinear form F is differentiable in u, then the entries of the Jacobian A are given by

$$A_{ij}(u_h) = \frac{\partial b_i(U)}{\partial U_j} = \frac{\partial}{\partial U_j} F(u_h; \hat{\phi}_i) = F'(u_h; \hat{\phi}_i) \, \frac{\partial u_h}{\partial U_j} = F'(u_h; \hat{\phi}_i) \, \phi_j \equiv F'(u_h; \hat{\phi}_i, \phi_j). \tag{3.13}$$

In each Newton iteration, we must then evaluate (assemble) the matrix A and the vector b, and update the solution vector U by

$$U^{k+1} = U^k - \delta U^k,$$

where δU^k solves the linear system

$$A(u_h^k) \, \delta U^k = b(u_h^k). \tag{3.14}$$

We note that for each fixed u_h, $a = F'(u_h; \cdot, \cdot)$ is a bilinear form and $L = F(u_h; \cdot)$ is a linear form. In each Newton iteration, we thus solve a linear variational problem of the canonical form (3.6): Find $\delta u \in V_0$ such that

$$F'(u_h; v, \delta u) = F(u_h; v) \quad \forall v \in \hat{V}, \tag{3.15}$$

where $V_0 = \{v - w : v, w \in V\}$. Discretizing (3.15) as in Section 3.3.1, we recover the linear system (3.14).

Example 3.1 (Nonlinear Poisson equation) *As an example, consider the following nonlinear Poisson equation:*

$$\begin{aligned} -\nabla \cdot ((1 + u)\nabla u) &= f \quad \textit{in } \Omega, \\ u &= 0 \quad \textit{on } \partial\Omega. \end{aligned} \tag{3.16}$$

Multiplying (3.16) with a test function v and integrating by parts, we obtain

$$\int_\Omega \nabla v \cdot ((1+u)\nabla u)\,\mathrm{d}x = \int_\Omega v f\,\mathrm{d}x,$$

which is a nonlinear variational problem of the form (3.10), with

$$F(u;v) = \int_\Omega \nabla v \cdot ((1+u)\nabla u)\,\mathrm{d}x - \int_\Omega v\,f\,\mathrm{d}x.$$

Linearizing the semilinear form F around $u = u_h$, we obtain

$$F'(u_h;v,\delta u) = \int_\Omega \nabla v \cdot (\delta u \nabla u_h)\,\mathrm{d}x + \int_\Omega \nabla v \cdot ((1+u_h)\nabla \delta u)\,\mathrm{d}x.$$

We may thus compute the entries of the Jacobian matrix $A(u_h)$ by

$$A_{ij}(u_h) = F'(u_h;\hat{\phi}_i,\phi_j) = \int_\Omega \nabla\hat{\phi}_i \cdot (\phi_j \nabla u_h)\,\mathrm{d}x + \int_\Omega \nabla\hat{\phi}_i \cdot ((1+u_h)\nabla\phi_j)\,\mathrm{d}x. \quad \textbf{(3.17)}$$

3.4 Finite Element Function Spaces

In the above discussion, we assumed that we could construct discrete subspaces $V_h \subset V$ of infinite dimensional function spaces. A central aspect of the finite element method is the construction of such subspaces by patching together local function spaces defined on a set of *finite elements*. We here give a general overview of the construction of finite element function spaces and return below in Chapters 4 and 5 to the construction of specific function spaces such as subsets of $H^1(\Omega)$, $H(\mathrm{curl})$, $H(\mathrm{div})$ and $L^2(\Omega)$.

3.4.1 The mesh

To define V_h, we first partition the domain Ω into a finite set of disjoint cells $\mathcal{T} = \{K\}$ such that

$$\cup_{K\in\mathcal{T}} K = \Omega.$$

Together, these cells form a *mesh* of the domain Ω. The cells are typically simple polygonal shapes like intervals, triangles, quadrilaterals, tetrahedra or hexahedra as shown in Figure 3.3. But other shapes are possible, in particular curved cells to correctly capture the boundary of a non-polygonal domain as shown in Figure 3.4.

missing figure to be added

Figure 3.3: Finite element cells in one, two and three space dimensions.

missing figure to be added

Figure 3.4: A straight triangular cell (left) and curved triangular cell (right).

3.4.2 The finite element definition

Once a domain Ω has been partitioned into cells, one may define a local function space $\mathcal{P}_K$ on each cell K and use these local function spaces to build the global function space V_h. A cell K together with a local function space $\mathcal{P}_K$ and a set of rules for describing functions in $\mathcal{P}_K$ is called a *finite element*. This definition of finite element was first formalized by Ciarlet in Ciarlet (1978, 2002a), and it remains the standard formulation today (Brenner and Scott, 1994, 2008). The formal definition reads as follows: A finite element is a triple $(K, \mathcal{P}_K, \mathcal{L}_K)$, where

- $K \subset \mathbb{R}^d$ is a bounded closed subset of $\mathbb{R}^d$ with nonempty interior and piecewise smooth boundary;
- $\mathcal{P}_K$ is a function space on K of dimension $n_K < \infty$;
- $\mathcal{L}_K = \{\ell_1^K, \ell_2^K, \dots, \ell_{n_K}^K\}$ is a basis for $\mathcal{P}_K'$ (the bounded linear functionals on $\mathcal{P}_K$).

As an example, consider the standard linear Lagrange finite element on the triangle in Figure 3.5. The cell K is given by the triangle and the space $\mathcal{P}_K$ is given by the space of first degree polynomials on K. As a basis for $\mathcal{P}_K'$, we may take point evaluation at the three vertices of K, that is,

$$\ell_i^K : \mathcal{P}_K \to \mathbb{R},$$
$$\ell_i^K(v) = v(x^i),$$

for $i = 1, 2, 3$ where x^i is the coordinate of the ith vertex. To check that this is indeed a finite element, we need to verify that $\mathcal{L}_K$ is a basis for $\mathcal{P}_K'$. This is equivalent to the unisolvence of $\mathcal{L}_K$, that is, if $v \in \mathcal{P}_K$ and $\ell_i^K(v) = 0$ for all ℓ_i^K, then $v = 0$. (Brenner and Scott, 2008) For the linear Lagrange triangle, we note that if v is zero at each vertex, then v must be zero everywhere, since a plane is uniquely determined by its value a three non-collinear points. Thus, the linear Lagrange triangle is indeed a finite element. In general, determining the unisolvence of $\mathcal{L}_K$ may be non-trivial.

3.4.3 The nodal basis

Expressing finite element solutions in V_h in terms of basis functions for the local function spaces $\mathcal{P}_K$ may be greatly simplified by introducing a *nodal basis* for $\mathcal{P}_K$. A nodal basis $\{\phi_i^K\}_{i=1}^{n_K}$ for $\mathcal{P}_K$ is a basis for $\mathcal{P}_K$ that satisfies

$$\ell_i^K(\phi_j^K) = \delta_{ij}, \quad i, j = 1, 2, \dots, n_K. \tag{3.18}$$

It follows that any $v \in \mathcal{P}_K$ may be expressed by

$$v = \sum_{i=1}^{n_K} \ell_i^K(v)\phi_i^K. \tag{3.19}$$

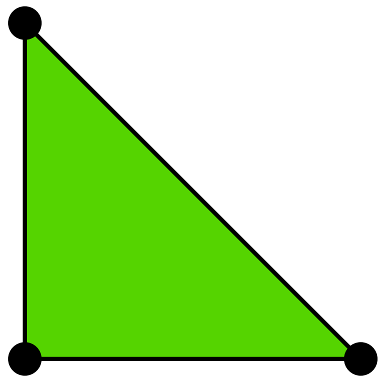


Figure 3.5: The linear Lagrange (Courant) triangle).

In particular, any function v in $\mathcal{P}_K$ for the linear Lagrange triangle is given by $v = \sum_{i=1}^{3} v(x^i)\phi_i^K$. In other words, the degrees of freedom of any function v may be obtained by evaluating the linear functionals $\mathcal{L}_K$. We shall therefore sometimes refer to $\mathcal{L}_K$ as degrees of freedom.
► Author note*: Give explicit formulas for some nodal basis functions. Use example environment.*

For any finite element $(K, \mathcal{P}_K, \mathcal{L}_K)$, the nodal basis may be computed by solving a linear system of size $n_K \times n_K$. To see this, let $\{\psi_i^K\}_{i=1}^{n_K}$ be any basis (the *prime* basis) for $\mathcal{P}_K$. Such a basis is easy to construct if $\mathcal{P}_K$ is a full polynomial space or may otherwise be computed by a singular-value decomposition or a Gram-Schmidt procedure, see (Kirby, 2004). We may then make an ansatz for the nodal basis in terms of the prime basis:

$$\phi_j = \sum_{k=1}^{n_K} \alpha_{jk}\psi_k^K, \quad j = 1, 2, \ldots, n_K.$$

Inserting this into (3.18), we find that

$$\sum_{k=1}^{n_K} \alpha_{jk}\ell_i^K(\psi_k^K) = \delta_{ij}, \quad j = 1, 2, \ldots, n_K.$$

In other words, the expansion coefficients α for the nodal basis may be computed by solving the linear system

$$B\alpha^\top = I,$$

where $B_{ij} = \ell_i^K(\psi_j^K)$.

3.4.4 The local-to-global mapping

Now, to define a global function space $V_h = \mathrm{span}\{\phi_i\}_{i=1}^{N}$ on Ω from a given set $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K\in\mathcal{T}}$ of finite elements, we also need to specify how the local function

missing figure to be added

Figure 3.6: Local-to-global mapping for a simple mesh consisting of two triangles.

spaces are patched together. We do this by specifying for each cell $K \in \mathcal{T}$ a *local-to-global mapping*,

$$\iota_K : [1, n_K] \to N. \tag{3.20}$$

This mapping specifies how the local degrees of freedom $\mathcal{L}_K = \{\ell_i^K\}_{i=1}^{n_K}$ are mapped to global degrees of freedom $\mathcal{L} = \{\ell_i\}_{i=1}^{N}$. More precisely, the global degrees of freedom are given by

$$\ell_{\iota_K(i)}(v) = \ell_i^K(v|_K), \quad i = 1, 2, \ldots, n_K, \tag{3.21}$$

for any $v \in V_h$. Thus, each local degree of freedom $\ell_i^K \in \mathcal{L}_K$ corresponds to a global degree of freedom $\nu_{\iota_K(i)} \in \mathcal{L}$ determined by the local-to-global mapping ι_K. As we shall see, the local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h.

For standard piecewise linears, one may define the local-to-global mapping by simply mapping each local vertex number i for $i = 1, 2, 3$ to the corresponding global vertex number $\iota_K(i)$. This is illustrated in Figure 3.6 for a simple mesh consisting of two triangles.

3.4.5 The global function space

One may now define the global function space V_h as the set of functions on Ω satisfying the following pair of conditions. We first require that

$$v|_K \in \mathcal{P}_K \quad \forall K \in \mathcal{T}, \tag{3.22}$$

that is, the restriction of v to each cell K lies in the local function space $\mathcal{P}_K$. Second, we require that for any pair of cells $(K, K') \in \mathcal{T} \times \mathcal{T}$ and any pair $(i, i') \in$

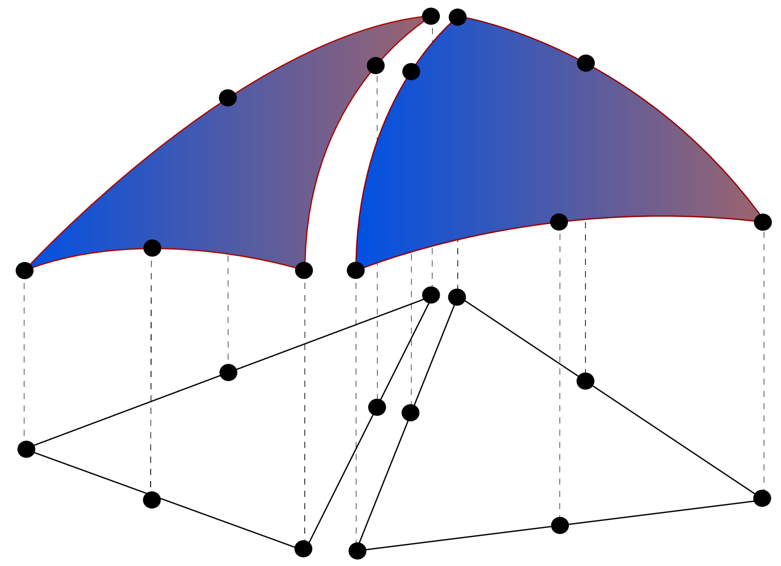


Figure 3.7: Patching together local function spaces on a pair of cells (K, K') to form a global function space on $\Omega = K \cup K'$.

$[1, n_K] \times [1, n_{K'}]$ satisfying

$$\iota_K(i) = \iota_{K'}(i'), \tag{3.23}$$

it holds that

$$\ell_i^K(v|_K) = \ell_{i'}^{K'}(v|_{K'}). \tag{3.24}$$

In other words, if two local degrees of freedom ℓ_i^K and $\ell_{i'}^{K'}$ are mapped to the same global degree of freedom, then they must agree for each function $v \in V_h$. Here, $v|_K$ denotes the continuous extension to K of the restriction of v to the interior of K. This is illustrated in Figure 3.7 for the space of continuous piecewise quadratics obtained by patching together two quadratic Lagrange triangles.

Note that by this construction, the functions of V_h are undefined on cell boundaries, unless the constraints (3.24) force the (restrictions of) functions of V_h to be continuous on cell boundaries. However, this is usually not a problem, since we can perform all operations on the restrictions of functions to the local cells.

The local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h. For the Lagrange triangle, the choice of degrees of freedom as point evaluation at vertices ensures that the restrictions $v|K$ and $v|K'$ of a function $v \in V_h$ to a pair of adjacent triangles K agree at the two common vertices, since ι_K and $\iota_{K'}$ map corresponding degrees of freedom to the same global degree of freedom and this global degree of freedom is single-valued. It follows that the functions of V_h are continuous not only at vertices but along each shared edge since a first-degree polynomial on a line is uniquely determined by its values at two distinct points. Thus, the global function space of piecewise linears generated by the Lagrange triangle is continuous

missing figure to be added

Figure 3.8: The degree of continuity is determined by the choice of degrees of freedom, illustrated here for a pair of linear Lagrange triangles, Crouzeix–Raviart triangles, Brezzi–Douglas–Marini triangles and Nédélec triangles.

and thus H^1-conforming, that is, $V_h \subset H^1(\Omega)$.

One may also consider degrees of freedom defined by point evaluation at the midpoint of each edge. This is the so-called Crouzeix–Raviart triangle. The corresponding global Crouzeix–Raviart space V_h is consequently continuous only at edge midpoints and so V_h is not a subspace of H^1. The Crouzeix–Raviart triangle is an example of an H^1-nonconforming element. Other choices of degrees of freedom may ensure continuity of normal components like for the $H(\mathrm{div})$-conforming Brezzi–Douglas–Marini elements or tangential components as for the $H(\mathrm{curl})$-conforming Nédélec elements. This is illustrated in Figure 3.8. In Chapter 4, other examples of particular elements are given which ensure different kinds of continuity by the choice of degrees of freedom and local-to-global mapping.

3.4.6 The mapping from the reference element

► Editor note*: Need to change the notation here to* $(K_0, \mathcal{P}_0, \mathcal{L}_0)$ *since* $\hat{\cdot}$ *is already used for the test space.*

As we have seen, the global function space V_h may be described by a mesh $\mathcal{T}$, a set of finite elements $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K\in\mathcal{T}}$ and a set of local-to-global mappings $\{\iota_K\}_{K\in\mathcal{T}}$. We may simplify this description further by introducing a *reference finite element* $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})$, where $\hat{\mathcal{L}} = \{\hat{\ell}_1, \hat{\ell}_2, \dots, \hat{\ell}_{\hat{n}}\}$, and a set of invertible mappings $\{F_K\}_{K\in\mathcal{T}}$ that map the reference cell $\hat{K}$ to the cells of the mesh,

$$K = F_K(\hat{K}) \quad \forall K \in \mathcal{T}. \tag{3.25}$$

This is illustrated in Figure 3.9. Note that $\hat{K}$ is generally not part of the mesh.

missing figure to be added

Figure 3.9: The (affine) mapping F_K from a reference cell $\hat{K}$ to some cell $K \in \mathcal{T}$.

For function spaces discretizing H^1 as in (3.3), the mapping F_K is typically *affine*, that is, F_K can be written in the form $F_K(\hat{x}) = A_K\hat{x} + b_K$ for some matrix $A_K \in \mathbb{R}^{d\times d}$ and some vector $b_K \in \mathbb{R}^d$, or *isoparametric*, in which case the components of F_K are functions in $\hat{\mathcal{P}}$. For function spaces discretizing $H(\mathrm{div})$ like in (3.5) or $H(\mathrm{curl})$, the appropriate mappings are the contravariant and covariant Piola mappings which preserve normal and tangential components respectively, see (Rognes et al., 2008). For simplicity, we restrict the following discussion to the case when F_K is affine or isoparametric.

For each cell $K \in \mathcal{T}$, the mapping F_K generates a function space on K given by

$$\mathcal{P}_K = \{v : v = \hat{v} \circ F_K^{-1}, \hat{v} \in \hat{\mathcal{P}}\}, \tag{3.26}$$

that is, each function $v = v(x)$ may be expressed as $v(x) = \hat{v}(F_K^{-1}(x)) = \hat{v} \circ F_K^{-1}(x)$ for some $\hat{v} \in \hat{\mathcal{P}}$.

The mapping F_K also generates a set of degrees of freedom $\mathcal{L}_K$ on $\mathcal{P}_K$ given by

$$\mathcal{L}_K = \{\ell_i^K : \ell_i^K(v) = \hat{\ell}_i(v \circ F_K), \quad i = 1, 2, \ldots, \hat{n}\}. \tag{3.27}$$

The mappings $\{F_K\}_{K\in\mathcal{T}}$ thus generate from the reference finite element $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})$ a set of finite elements $\{(K, \mathcal{P}_K, \mathcal{L}_K)\}_{K\in\mathcal{T}}$ given by

$$\begin{aligned} K &= F_K(\hat{K}), \\ \mathcal{P}_K &= \{v : v = \hat{v} \circ F_K^{-1} : \hat{v} \in \hat{\mathcal{P}}\}, \\ \mathcal{L}_K &= \{\ell_i^K : \ell_i^K(v) = \hat{\ell}_i(v \circ F_K), \quad i = 1, 2, \ldots, \hat{n} = n_K\}. \end{aligned} \tag{3.28}$$

By this construction, we also obtain the nodal basis functions $\{\phi_i^K\}_{i=1}^{n_K}$ on K from a set of nodal basis functions $\{\hat{\phi}_i\}_{i=1}^{\hat{n}}$ on the reference element satisfying $\hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}$.

Letting $\phi_i^K = \hat{\phi}_i \circ F_K^{-1}$ for $i = 1, 2, \ldots, n_K$, we find that

$$\ell_i^K(\phi_j^K) = \hat{\ell}_i(\phi_j^K \circ F_K) = \hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}, \tag{3.29}$$

so $\{\phi_i^K\}_{i=1}^{n_K}$ is a nodal basis for $\mathcal{P}_K$.

We may thus define the function space V_h by specifying a mesh $\mathcal{T}$, a reference finite element $(\hat{K}, \hat{\mathcal{P}}, \hat{\mathcal{L}})$, a set of local-to-global mappings $\{\iota_K\}_{K\in\mathcal{T}}$ and a set of mappings $\{F_K\}_{K\in\mathcal{T}}$ from the reference cell $\hat{K}$. Note that in general, the mappings need not be of the same type for all cells K and not all finite elements need to be generated from the same reference finite element. In particular, one could employ a different (higher-degree) isoparametric mapping for cells on a curved boundary.

The above construction is valid for so-called affine-equivalent elements (Brenner and Scott, 2008) like the family H^1-conforming Lagrange finite elements. A similar construction is possible for $H(\mathrm{div})$- and $H(\mathrm{curl})$ conforming elements, like the Raviart–Thomas, Brezzi–Douglas–Marini and Nédélec elements, where an appropriate Piola mapping must be used to map the basis functions. However, not all finite elements may be generated from a reference finite element using this simple construction. For example, this construction fails for the family of Hermite finite elements. (Brenner and Scott, 2008, Ciarlet, 2002a).

3.5 Finite Element Solvers

Finite elements provide a powerful methodology for discretizing differential equations, but solving the resulting algebraic systems also presents quite a challenge, even for linear systems. Good solvers must handle the sparsity and ill-conditioning of the algebraic system, but also scale well on parallel computers. The linear solve is a fundamental operation not only in linear problems, but also within each iteration of a nonlinear solve via Newton's method, an eigenvalue solve, or time-stepping.

A classical approach that has been revived recently is direct solution, based on Gaussian elimination. Thanks to techniques enabling parallel scalability and recognizing block structure, packages such as UMFPACK (Davis, 2004) and SuperLU (Li, 2005) have made direct methods competitive for quite large problems.

The 1970s and 1980s saw the advent of modern iterative methods. These grew out of classical iterative methods such as relaxation methods (**?**) and the conjugate gradient iteration of Hestenes and Stieffel (Hestenes and Stiefel, 1952). These techniques can use much less memory than direct methods and are easier to parallelize.

► Author note*: Missing reference for relaxation methods*

Multigrid methods (Brandt, 1977, Wesseling, 1992) use relaxation techniques on a hierarchy of meshes to solve elliptic equations, typically for symmetric problems, in nearly linear time. However, they require a hierarchy of meshes that may not always be available. This motivated the introduction of *algebraic* multigrid methods (AMG) that mimic mesh coarsening, working only on the matrix entries. Successful AMG distributions include the Hypre package (Falgout and Yang, 2002) and the ML package inside Trilinos (Heroux et al., 2005).

Krylov methods such as conjugate gradients and GMRES (Saad and Schultz, 1986) generate a sequence of approximations converging to the solution of the linear system. These methods are based only on the matrix–vector product. The performance of these methods is significantly improved by use of *preconditioners*, which transform the linear system

$$AU = b$$

into

$$P^{-1}AU = P^{-1}b,$$

which is known as left preconditioning. The preconditioner P^{-1} may also be applied from the right by recognizing that $AU = AP^{-1}(PU)$. To ensure good convergence, the preconditioner P^{-1} should be a good approximation of A^{-1}. Some preconditioners are strictly algebraic, meaning they only use information available from the entries of A. Classical relaxation methods such as Gauss–Seidel may be used as preconditioners, as can so-called incomplete factorizations (**?**). If multigrid or AMG is available, it also can serve as a powerful preconditioner. Other kinds of preconditioners require special knowledge about the differential equations being solved and may require new matrices modeling related physical processes. Such methods are sometimes called *physics-based* preconditioners (**?**). An automated system, such as FEniCS, provides an interesting opportunity to assist with the development and implementation of these powerful but less widely used methods.

► Author note*: Missing reference for incomplete LU factorization and physics-based preconditioners*

Fortunately, many of the methods discussed here are included in modern libraries such as PETSc (Balay et al., 2004) and Trilinos (Heroux et al., 2005). FEniCS typically interacts with the solvers discussed here through these packages and so mainly need to be aware of the various methods at a high level, such as when the various methods are appropriate and how to access them.

3.6 Finite Element Error Estimation and Adaptivity

The error $e = u_h - u$ in a computed finite element solution u_h approximating the exact solution u of (3.6) may be estimated either *a priori* or *a posteriori*. Both types of estimates are based on relating the size of the error to the size of the (weak) residual $r : V \to \mathbb{R}$ defined by

$$r(v) = a(v, u_h) - L(v). \tag{3.30}$$

We note that the weak residual is formally related to the strong residual $R \in V'$ by $r(v) = (v, R)$.

A priori error estimates express the error in terms of the regularity of the exact (unknown) solution and may give useful information about the order of convergence of a finite element method. A posteriori error estimates express the error in terms of computable quantities like the residual and (possibly) the solution of an auxiliary dual problem.

3.6.1 A priori error analysis

We consider the linear variational problem (3.6). We first assume that the bilinear form a and the linear form L are continuous (bounded), that is, there exists a constant $C > 0$ such that

$$\begin{aligned} a(v, w) &\leq C\|v\|_V\|w\|_V, & (3.31)\\ L(v) &\leq C\|v\|_V, & (3.32)\end{aligned}$$

for all $v, w \in V$. For simplicity, we assume in this section that $\hat{V} = V$ is a Hilbert space. For (3.1), this corresponds to the case of homogeneous Dirichlet boundary conditions and $V = H_0^1(\Omega)$. Extensions to the general case $\hat{V} \neq V$ are possible, see for example (Oden and Demkowicz, 1996). We further assume that the bilinear form a is coercive (V-elliptic), that is, there exists a constant $\alpha > 0$ such that

$$a(v, v) \geq \alpha\|v\|_V, \tag{3.33}$$

for all $v \in V$. It then follows by the Lax–Milgram theorem (Lax and Milgram, 1954) that there exists a unique solution $u \in V$ to the variational problem (3.6).

To derive an a priori error estimate for the approximate solution u_h defined by the discrete variational problem (3.7), we first note that

$$a(v, u_h - u) = a(v, u_h) - a(v, u) = L(v) - L(v) = 0$$

for all $v \in V_h \subset V$. By the coercivity and continuity of the bilinear form a, we find that

$$\begin{aligned}\alpha\|u_h - u\|_V^2 \leq a(u_h - u, u_h - u) &= a(u_h - v, u_h - u) + a(v - u, u_h - u)\\ &= a(v - u, u_h - u) \leq C\|v - u\|_V \, \|u_h - u\|_V.\end{aligned}$$

for all $v \in V_h$. It follows that

$$\|u_h - u\|_V \leq \frac{C}{\alpha}\|v - u\|_V \quad \forall v \in V_h. \tag{3.34}$$

missing figure to be added

Figure 3.10: The finite element solution $u_h \in V_h \subset V$ is the a-projection of $u \in V$ onto the subspace V_h and is consequently the best possible approximation of u in the subspace V_h.

The estimate (3.34) is referred to as Cea's lemma. We note that when the bilinear form a is symmetric, it is also an inner product. We may then take $\|v\|_V = \sqrt{a(v,v)}$ and $C = \alpha = 1$. In this case, u_h is the a-projection onto V_h and Cea's lemma states that

$$\|u_h - u\|_V \leq \|v - u\|_V \quad \forall v \in V_h, \tag{3.35}$$

that is, u_h is the best possible solution of the variational problem (3.6) in the subspace V_h. This is illustrated in Figure 3.10.

Cea's lemma together with a suitable interpolation estimate now yields the a priori error estimate for u_h. By choosing $v = \pi_h u$, where $\pi_h : V \to V_h$ is an interpolation operator into V_h, we find that

$$\|u_h - u\|_V \leq \frac{C}{\alpha}\|\pi_h u - u\|_V \leq \frac{CC_i}{\alpha}\|h^p D^{q+1} u\|, \tag{3.36}$$

where C_i is an interpolation constant and the values of p and q depend on the accuracy of interpolation and the definition of $\|\cdot\|_V$. For the solution of Poisson's equation in H_0^1, we have $C = \alpha = 1$ and $p = q = 1$.

3.6.2 A posteriori error analysis

Energy norm error estimates

The continuity and coercivity of the bilinear form a also allows a simple derivation of an a posteriori error estimate. In fact, it follows that the V-norm of the error $e = u_h - u$ is equivalent to the V'-norm of the residual r. To see this, we note that by the continuity of the bilinear form a, we have

$$r(v) = a(v, u_h) - L(v) = a(v, u_h) - a(v, u) = a(v, u_h - u) \leq C\|u_h - u\|_V \, \|v\|_V.$$

Furthermore, by coercivity, we find that

$$\alpha\|u_h - u\|_V^2 \leq a(u_h - u, u_h - u) = a(u_h - u, u_h) - L(u_h - u) = r(u_h - u).$$

It follows that

$$\alpha\|u_h - u\|_V \leq \|r\|_{V'} \leq C\|u_h - u\|_V, \tag{3.37}$$

where $\|r\|_{V'} = \sup_{v \in V, v \neq 0} r(v)/\|v\|_V$.

The estimates (3.36) and (3.37) are sometimes referred to as *energy norm* error estimates. This is the case when the bilinear form a is symmetric and thus defines an inner product. One may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, it follows that

$$\|e\|_V = \|r\|_{V'}. \tag{3.38}$$

The term energy norm refers to $a(v, v)$ corresponding to physical energy in many applications.

Duality-based error control

The classical a priori and a posteriori error estimates (3.36) and (3.37) relate the V-norm of the error $e = u_h - u$ to the regularity of the exact solution u and the residual $r = a(v, u_u) - L(v)$ of the finite element solution u_h respectively. However, in applications it is often necessary to control the error in a certain *output functional* $\mathcal{M} : V \to \mathbb{R}$ of the computed solution to within some given tolerance $\mathrm{TOL} > 0$. In these situations, one would thus ideally like to choose the finite element space $V_h \subset V$ such that the finite element solution u_h satisfies

$$|\mathcal{M}(u_h) - \mathcal{M}(u)| \leq \mathrm{TOL} \tag{3.39}$$

with minimal computational work. We assume here that both the output functional and the variational problem are linear, but the analysis may be easily extended to the full nonlinear case, see (Becker and Rannacher, 2001, Eriksson et al., 1995).

To estimate the error in the output functional $\mathcal{M}$, we introduce an auxiliary *dual* problem: Find $z \in V^*$ such that

$$a^*(v, z) = \mathcal{M}(v) \quad \forall v \in \hat{V}^*. \tag{3.40}$$

We note here that the functional $\mathcal{M}$ enters as data in the dual problem. The dual (adjoint) bilinear form $a^* : \hat{V}^* \times V^*$ is defined by

$$a^*(v, w) = a(w, v).$$

The dual trial and test spaces are given by

$$\begin{aligned} V^* &= \hat{V}, \\ \hat{V}^* &= V_0 = \{v - w : v, w \in V\}, \end{aligned}$$

that is, the dual trial space is the primal test space and the dual test space is the primal trial space modulo boundary conditions. In particular, if $V = u_0 + \hat{V}$ and $V_h = u_0 + \hat{V}_h$ then $\hat{V}^* = \hat{V}$, and thus both the dual test and trial functions vanish at Dirichlet boundaries. The definition of the dual problem leads us to the following representation of the error:

$$\begin{aligned} \mathcal{M}(u_h) - \mathcal{M}(u) &= \mathcal{M}(u_h - u) \\ &= a^*(u_h - u, z) \\ &= a(z, u_h - u) \\ &= a(z, u_h) - L(z) \\ &= r(z). \end{aligned}$$

We thus find that the error is exactly represented by the residual of the dual solution,

$$\mathcal{M}(u_h) - \mathcal{M}(u) = r(z). \tag{3.41}$$

3.6.3 Adaptivity

As seen above, one may thus estimate the error in a computed finite element solution u_h, either the error in the V-norm or the error in an output functional, by estimating the size of the residual r. This may be done in several different ways. The estimate typically involves integration by parts to recover the strong element-wise residual of the original PDE, possibly in combination with the solution of local problems over cells or patches of cells. In the case of the standard piecewise linear finite element approximation of Poisson's equation (3.1), one may obtain the following estimate:

$$\|u_h - u\|_V = \|\nabla e\| \leq C \left(\sum_{K \in \mathcal{T}} h_K^2 \|R\|_K^2 + h_K \|[\partial_n u_h]\|_{\partial K}^2 \right)^{1/2},$$

where $R|_K = -\Delta u_h|_K - f|_K$ is the strong residual, h_K denotes the mesh size (diameter of smallest circumscribed sphere) and $[\partial_n u_h]$ denotes the jump of the normal derivative across mesh facets. Letting $\eta_K^2 = h_K^2 \|R\|_K^2 + h_K \|[\partial_n u_h]\|_{\partial K}^2$, one thus obtains the estimate

$$\|u_h - u\|_V \leq E \equiv \left(C \sum_K \eta_K^2 \right)^{1/2}.$$

missing figure to be added

Figure 3.11: An adaptively refined mesh obtained by successive refinement of an original coarse mesh.

An adaptive algorithm seeks to determine a mesh size $h = h(x)$ such that $E \leq \mathrm{TOL}$. Starting from an initial coarse mesh, the mesh is successively refined in those cells where the error indicator η_K is large. Several strategies are available, such as refining the top fraction of all cells where η_K is large, say the first 20% of all cells ordered by η_K. Other strategies include refining all cells where η_K is above a certain fraction of $\max_{K \in \mathcal{T}} \eta_K$, or refining a top fraction of all cells such that the sum of their error indicators account for a significant fraction of E.
► Author note*: Find good reference for adaptive strategies.*

Once the mesh has been refined, a new solution and new error indicators can be computed. The process is then repeated until either $E \leq \mathrm{TOL}$ (the stopping criterion) or the available resources (CPU time and memory) have been exhausted. The adaptive algorithm thus yields a sequence of successively refined meshes as illustrated in Figure 3.11. For time-dependent problems, an adaptive algorithm needs to distribute both the mesh size and the size of the time step in both space and time. Ideally, the error estimate E is close to the actual error, as measured by the *efficiency index* $E/\|u_h - u\|_V$ which should be close to one by bounded below by one.

3.7 Automating the Finite Element Method

The FEniCS project seeks to automate Scientific Computing as explained in Chapter [intro]. This is a formidable task, but it may be solved in part by automating the finite element method. In particular, this automation relies on the following key steps:

(i) automation of discretization,

(ii) automation of discrete solution,

(iii) automation of error control.

Since its inception in 2003, the FEniCS project has been concerned mainly with the automation of discretization, resulting in the development of the form compilers FFC and SyFi/SFC, the code generation interface UFC and the form language UFL. As a result, the first step towards a complete automation is now close to complete; variational problems for a large class of partial differential equations may now be automatically discretized by the finite element method using FEniCS. For the automation of discrete solution, that is, the solution of linear and nonlinear systems arising from the automated discretization of variational problems, interfaces to state-of-the-art libraries for linear algebra have been implemented as part of DOLFIN. Ongoing work is now seeking to automate error control by automated error estimation and adaptivity as part of FEniCS.

3.8 Outlook

In the following chapters, we return to specific aspects of the automation of the finite element method. In the next chapter, we review a number of common and unusual finite elements, including the standard Lagrange elements but also some more exotic elements. In Chapter 5, we then discuss the automated generation of finite element nodal basis functions from a given finite element definition $(K, \mathcal{P}_K, \mathcal{L}_K)$. In Chapter 6, we consider general finite element variational forms arising from the discretization of PDEs and discuss the automated assembly of the corresponding discrete operators in Chapter 7. We then discuss specific optimization strategies for form evaluation in Chapters **??**–**??**.
▶ Author note*: This section needs to be reworked when the following chapters have materialized.*

3.9 Historical Notes

In 1915, Boris Grigoryevich Galerkin formulated a general method for solving differential equations. (Galerkin, 1915) A similar approach was presented sometime earlier by Bubnov. Galerkin's method, or the Bubnov–Galerkin method, was originally formulated with global polynomials and goes back to the variational principles of Leibniz, Euler, Lagrange, Dirichlet, Hamilton, Castigliano (Castigliano, 1879), Rayleigh (Rayleigh, 1870) and Ritz (Ritz, 1908). Galerkin's method with piecewise polynomial spaces $(\hat{V}_h, V_h)$ is known as the *finite element method*. The

Figure 3.12: Boris Galerkin (1871–1945), inventor of Galerkin's method.

finite element method was introduced by engineers for structural analysis in the 1950s and was independently proposed by Courant in 1943 (Courant, 1943). The exploitation of the finite element method among engineers and mathematicians exploded in the 1960s. Since then, the machinery of the finite element method has been expanded and refined into a comprehensive framework for design and analysis of numerical methods for differential equations, see (Becker et al., 1981, Brenner and Scott, 1994, Ciarlet, 1976, 1978, Hughes, 1987, Strang and Fix, 1973, Zienkiewicz et al., 2005, first published in 1967) Recently, the quest for compatible (stable) discretizations of mixed variational problems has led to the introduction of finite element exterior calculus. (Arnold et al., 2006a)

Work on a posteriori error analysis of finite element methods dates back to the pioneering work of Babuška and Rheinboldt. (Babuška and Rheinboldt, 1978). Important references include the works (Ainsworth and Oden, 1993, Bank and Weiser, 1985, Eriksson and Johnson, 1991, 1995a,b,c, Eriksson and Johnson, Eriksson et al., 1998, Zienkiewicz and Zhu, 1987) and the reviews papers (Ainsworth and Oden, 2000, Becker and Rannacher, 2001, Eriksson et al., 1995, Verfürth, 1994, 1999).

► Author note*: Need to check for missing/inaccurate references here.*

► Editor note*: Might use a special box/layout for historical notes if they appear in many places.*

CHAPTER 4

Common and Unusual Finite Elements

By Robert C. Kirby, Anders Logg and Andy R. Terrel

Chapter ref: **[kirby-6]**

This chapter provides a glimpse of the considerable range of finite elements in the literature and the challenges that may be involved with automating "all" the elements. Many of the elements presented here are included in the FEniCS project already; some are future work.

4.1 Ciarlet's Finite Element Definition

As discussed in Chapter 3, a finite element is defined by a triple $(K, \mathcal{P}_K, \mathcal{L}_K)$, where

- $K \subset \mathbb{R}^d$ is a bounded closed subset of $\mathbb{R}^d$ with nonempty interior and piecewise smooth boundary;
- $\mathcal{P}_K$ is a function space on K of dimension $n_K < \infty$;
- $\mathcal{L}_K = \{\ell_1^K, \ell_2^K, \ldots, \ell_{n_K}^K\}$ is a basis for $\mathcal{P}_K'$ (the bounded linear functionals on $\mathcal{P}_K$).

This definition was first introduced by Ciarlet in a set of lecture notes (Ciarlet, 1975) and became popular after his 1978 book (Ciarlet, 1978, 2002a). It remains the standard definition today, see for example (Brenner and Scott, 2008). Similar ideas were introduced earlier in (Ciarlet and Raviart, 1972) which discusses unisolvence of a set of interpolation points $\Sigma = \{a_i\}_i$. This is closely related to the unisolvence of $\mathcal{L}_K$. In fact, the set of functionals $\mathcal{L}_K$ is given by $\ell_i^K(v) = v(a_i)$. It is also interesting to note that the Ciarlet triple was originally written as (K, P, Σ) with Σ denoting $\mathcal{L}_K$. Conditions for uniquely determining a polynomial based on interpolation of function values and derivatives at a set of points was also discussed in (Bramble and Zlámal, 1970), although the term unisolvence was not used.

4.2 Notation

It is common to refer to the space of linear functionals $\mathcal{L}_K$ as the *degrees of freedom* of the element $(K, \mathcal{P}_K, \mathcal{L}_K)$. The degrees of freedom are typically given by point evaluation or moments of function values or derivatives. Other commonly used degrees of freedom are point evaluation or moments of certain components of function values, such as normal or tangential components, but also directional derivatives. We summarize the notation used to indicate degrees of freedom graphically in Figure 4.1. A filled circle at a point $\bar{x}$ denotes point evaluation at that point,

$$\ell(v) = v(\bar{x}).$$

We note that for a vector valued function v with d components, a filled circle denotes evaluation of all components and thus corresponds to d degrees of freedom,

$$\begin{aligned}\ell_1(v) &= v_1(\bar{x}),\\ \ell_2(v) &= v_2(\bar{x}),\\ \ell_3(v) &= v_3(\bar{x}).\end{aligned}$$

An arrow denotes evaluation of a component of a function value in a given direction, such as a normal component $\ell(v) = v(\bar{x}) \cdot n$ or tangential component $\ell(v) = v(\bar{x}) \cdot t$. A plain circle denotes evaluation of all first derivatives, a line denotes evaluation of a directional first derivative such as a normal derivative $\ell(v) = \nabla v(\bar{x}) \cdot n$. A dotted circle denotes evaluation of all second derivatives. Finally, a circle with a number indicates a number of interior moments (integration against functions over the domain K).

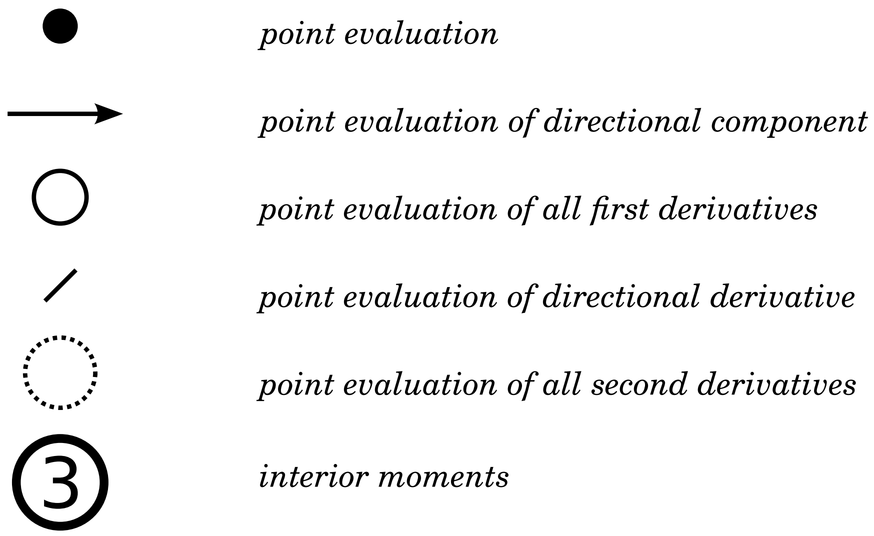

point evaluation
point evaluation of directional component
point evaluation of all first derivatives
point evaluation of directional derivative
point evaluation of all second derivatives
3
interior moments

Figure 4.1: Notation

4.3 The Argyris Element

4.3.1 Definition

The Argyris triangle (Argyris et al., 1968, Ciarlet, 2002a) is based on the space $\mathcal{P}_K = P_5(K)$ of quintic polynomials over some triangle K. It can be pieced together with full C^1 continuity between elements with C^2 continuity at the vertices of a triangulation. Quintic polynomials in $\mathbb{R}^2$ are a 21-dimensional space, and the dual basis $\mathcal{L}_K$ consists of six degrees of freedom per vertex and one per each edge. The vertex degrees of freedom are the function value, two first derivatives to specify the gradient, and three second derivatives to specify the unique components of the (symmetric) Hessian matrix.

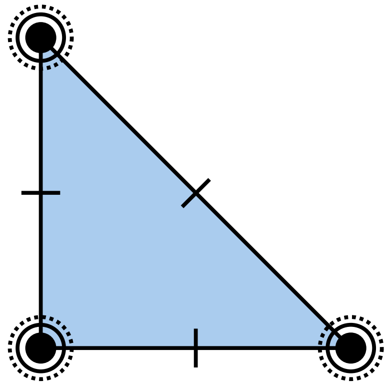


Figure 4.2: The quintic Argyris triangle.

4.3.2 Historical notes

The Argyris element (Argyris et al., 1968) was first called the TUBA element and was applied to fourth-order plate-bending problems. In fact, as Ciarlet points out (Ciarlet, 2002a), the element also appeared in an earlier work by Felippa (Felippa, 1966).

The normal derivatives in the dual basis for the Argyris element prevent it from being affine-interpolation equivalent. This prevents the nodal basis from being constructed on a reference cell and affinely mapped. Recent work by Dominguez and Sayas (Domínguez and Sayas, 2008) has developed a transformation that corrects this issue and requires less computational effort than directly forming the basis on each cell in a mesh.

The Argyris element can be generalized to polynomial degrees higher than quintic, still giving C^1 continuity with C^2 continuity at the vertices (Šolín et al., 2004). The Argyris element also makes an appearance in exact sequences of finite elements, where differential complexes are used to explain the stability of many kinds of finite elements and derive new ones (Arnold et al., 2006a).

4.4 The Brezzi–Douglas–Marini element

4.4.1 Definition

The Brezzi–Douglas–Marini element (Brezzi et al., 1985a) discretizes $H(\mathrm{div})$. That is, it provides a vector field that may be assembled with continuous normal components so that global divergences are well-defined. The BDM space on a simplex in d dimensions ($d = 2, 3$) consists of vectors of length d whose components are polynomials of degree q for $q \geq 1$.

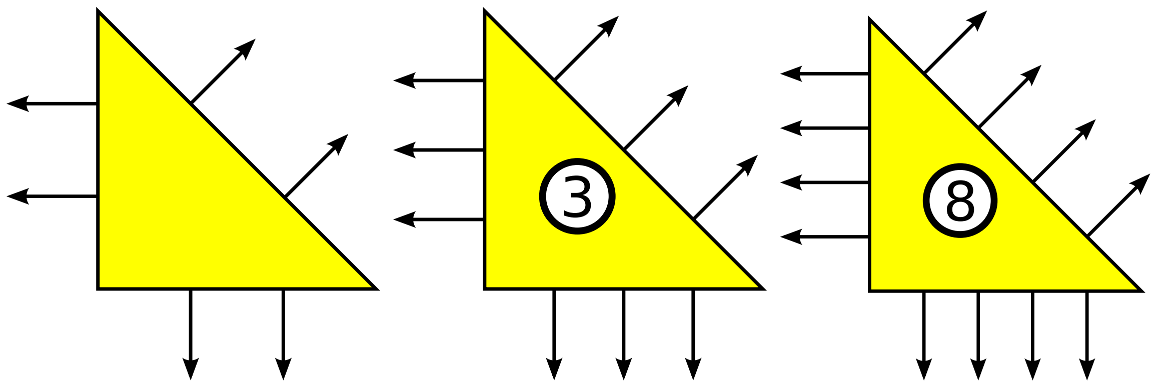


Figure 4.3: The linear, quadratic and cubic Brezzi–Douglas–Marini triangles.

The degrees of freedom for the BDM triangle include the normal component on each edge, specified either by integral moments against P_q or the value of the normal component at $q+1$ points per edge. For $q > 1$, the degrees of freedom also include integration against gradients of $P_q(K)$ over K. For $q > 2$, the degrees of freedom also include integration against curls of $b_K P_{q-2}(K)$ over K, where b_K is the cubic bubble function associated with K.

► Author note*: What about tets? Will also make up for the empty space on the next page.*

The BDM element is also defined on rectangles and boxes, although it has quite a different flavor. Unusually for rectangular domains, it is not defined using tensor products of one-dimensional polynomials, but instead by supplementing polynomials of complete degree $[P_q(K)]^d$ with extra functions to make the divergence onto $P_q(K)$. The boundary degrees of freedom are similar to the simplicial case, but the internal degrees of freedom are integral moments against $[P_q(K)]^d$.

4.4.2 Historical notes

The BDM element was originally derived in two dimensions (Brezzi et al., 1985a) as an alternative to the Raviart–Thomas element using a complete polynomial space. Extensions to tetrahedra came via the “second-kind” elements of Nédélec (Nédélec, 1986) as well as in Brezzi and Fortin (Brezzi and Fortin, 1991). While Nédélec uses quite different internal degrees of freedom (integral moments against the Raviart–Thomas spaces), the degrees of freedom in Brezzi and Fortin are quite similar to (Brezzi et al., 1985a).

A slight modification of the BDM element constrains the normal components on the boundary to be of degree $q-1$ rather than q. This is called the Brezzi–Douglas–Fortin–Marini or BDFM element (Brezzi and Fortin, 1991). In similar spirit, elements with differing orders on the boundary suitable for varying the polynomial degree between triangles were derived in (Brezzi et al., 1985b). Besides mixed formulations of second-order scalar elliptic equations, the BDM element also appears in elasticity (Arnold et al., 2007), where it is seen that each row of the stress tensor may be approximated in a BDM space with the symmetry of the stress tensor imposed weakly.

► Author note*: Fill up the blank space here. Adding a discussion and possibly a figure for tets should help.*

4.5 The Crouzeix–Raviart element

4.5.1 Definition

The Crouzeix–Raviart element (Crouzeix and Raviart, 1973) most commonly refers to a linear non-conforming element. It uses piecewise linear polynomials, but unlike the Lagrange element, the degrees of freedom are located at edge midpoints rather than at vertices. This gives rise to a weaker form of continuity, but it is still a suitable C^0-nonconforming element. The extension to tetrahedra in $\mathbb{R}^3$ replaces the degrees of freedom on edge midpoints by degrees of freedom on face midpoints.
▶ Author note: *What other element does it refer to? Sounds like there may be several, but I just know about this one.*

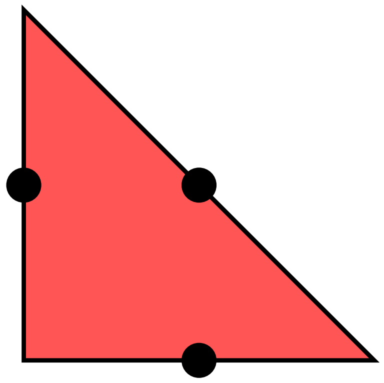


Figure 4.4: The linear Crouzeix–Raviart triangle.

4.5.2 Historical notes

Crouzeix and Raviart developed two simple Stokes elements, both using pointwise evaluation for degrees of freedom. The second element used extra bubble functions to enrich the typical Lagrange element, but the work of Crouzeix and Falk (Crouzeix and Falk, 1989) later showed that the bubble functions were in fact not necessary for quadratic and higher orders.
▶ Author note: *The discussion in the previous paragraph should be expanded so it states more explicitly what this has to do with the CR element.*

The element is usually associated with solving the Stokes problem but has been used for linear elasticity (Hansbo and Larson, 2003) and Reissner-Mindlin plates (Arnold and Falk, 1989) as a remedy for locking. There is an odd order extension of the element from Arnold and Falk.
▶ Author note: *Missing reference here to odd order extension.*

4.6 The Hermite Element

4.6.1 Definition

The Hermite element (Ciarlet, 2002a) generalizes the classic cubic Hermite interpolating polynomials on the line segment. On the triangle, the space of cubic polynomials is ten-dimensional, and the ten degrees of freedom are point evaluation at the triangle vertices and barycenter, together with the components of the gradient evaluated at the vertices. The generalization to tetrahedra is analagous.

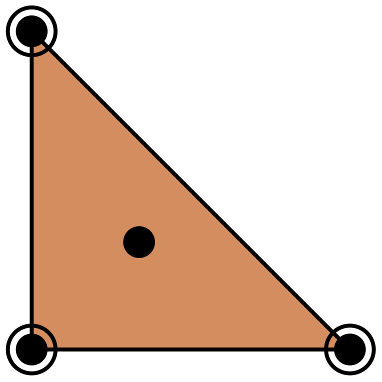


Figure 4.5: The cubic Hermite triangle.

Unlike the cubic Hermite functions on a line segment, the cubic Hermite triangle and tetrahedron cannot be patched together in a fully C^1 fashion.

4.6.2 Historical notes

Hermite-type elements appear in the finite element literature almost from the beginning, appearing at least as early as the classic paper of Ciarlet and Raviart (Ciarlet and Rav
1972). They have long been known as useful C^1-nonconforming elements (Braess, 2007, Ciarlet, 2002a). Under affine mapping, the Hermite elements form *affine-interpolation equivalent* families. (Brenner and Scott, 2008).

4.7 The Lagrange Element

4.7.1 Definition

The best-known and most widely used finite element is the Lagrange P_1 element. In general, the Lagrange element uses $\mathcal{P}_K = P_q(K)$, polynomials of degree q on K, and the degrees of freedom are simply pointwise evaluation at an array of points. While numerical conditioning and interpolation properties can be dramatically improved by choosing these points in a clever way (**?**), for the purposes of this chapter the points may be assumed to lie on an equispaced lattice.

▶ Author note*: Missing reference for statement about node placement.*

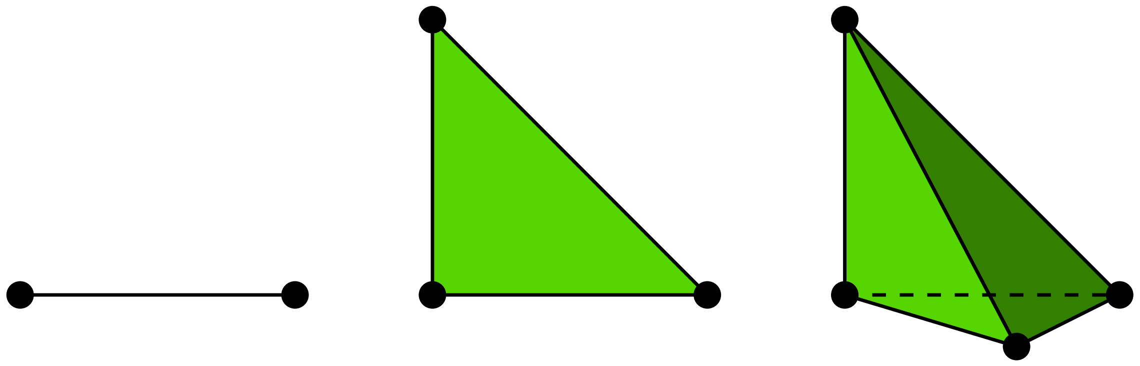


Figure 4.6: The linear Lagrange interval, triangle and tetrahedron.

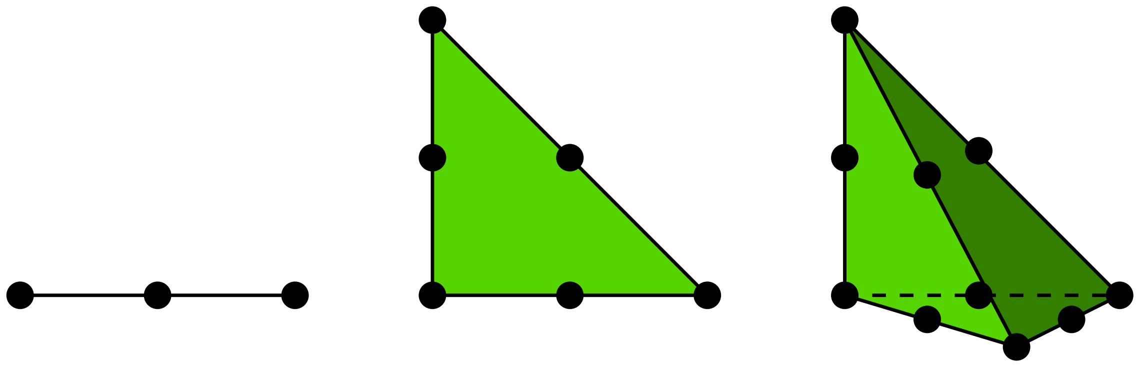


Figure 4.7: The quadratic Lagrange interval, triangle and tetrahedron.

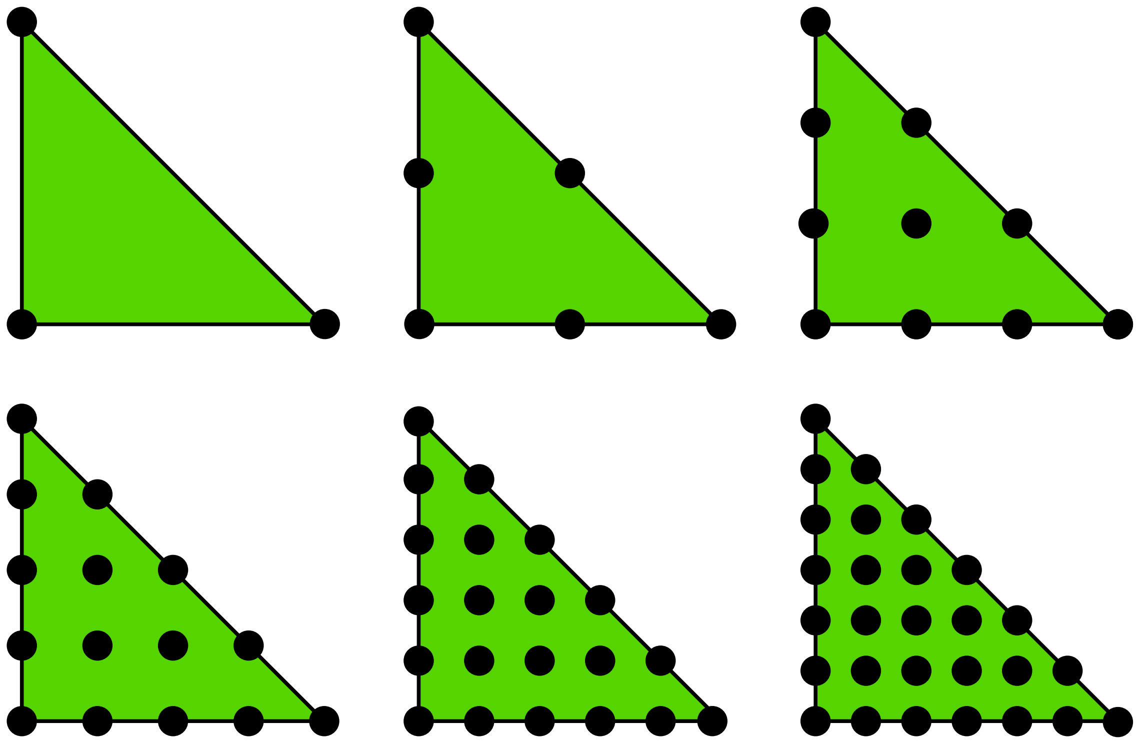


Figure 4.8: The Lagrange P_q triangle for $q = 1, 2, 3, 4, 5, 6$.

4.7.2 Historical notes

Reams could be filled with all the uses of the Lagrange elements. The Lagrange element predates the modern study of finite elements. The lowest-order triangle is sometimes called the *Courant* triangle, after the seminal paper (Courant, 1943) in which variational techniques are considered and the P_1 triangle is used to derive a finite difference method. The rest is history.

► Author note*: Expand the historical notes for the Lagrange element. As far as I can see, Bramble and Zlamal don't seem to be aware of the higher order Lagrange elements (only the Courant triangle). Their paper from 1970 focuses only on Hermite interpolation.*

4.8 The Morley Element

4.8.1 Definition

The Morley triangle (Morley, 1968) is a simple H^2-nonconforming quadratic element that is used in fourth-order problems. The function space is simply $\mathcal{P}_K = P_2(K)$, the six-dimensional space of quadratics. The degrees of freedom consist of pointwise evaluation at each vertex and the normal derivative at each edge midpoint. It is interesting that the Morley triangle is neither C^1 nor even C^0, yet it is suitable for fourth-order problems, and is the simplest known element for this purpose.

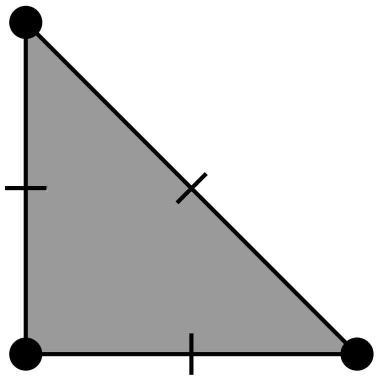


Figure 4.9: The quadratic Morley triangle.

4.8.2 Historical notes

The Morley element was first introduced to the engineering literature by Morley in 1968 (Morley, 1968). In the mathematical literature, Lascaux and Lesaint (Lascaux and Lesaint, 1975) considered it in the context of the patch test in a study of plate-bending elements.

► Author note: *Fill up page.*

4.9 The Nédélec Element

4.9.1 Definition

The widely celebrated $H(\mathrm{curl})$-conforming elements of Nédélec (Nédélec, 1980, 1986) are much used in electromagnetic calculations and stand as a premier example of the power of "nonstandard" (meaning not lowest-order Lagrange) finite elements.

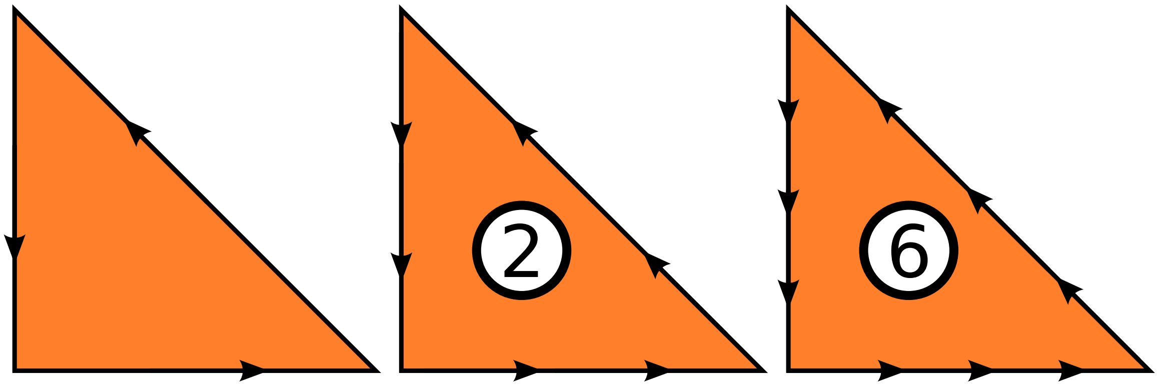


Figure 4.10: The linear, quadratic and cubic Nédélec triangles.

On triangles, the function space $\mathcal{P}_K$ may be obtained by a simple rotation of the Raviart–Thomas basis functions, but the construction of the tetrahedral element is substantially different. In the lowest order case $q = 1$, the space $\mathcal{P}_K$ may be written as functions of the form

$$v(x) = \alpha + \beta \times x,$$

where α and β are vectors in $\mathbb{R}^3$. Hence, $\mathcal{P}_K$ contains all vector-valued constant functions and some but not all linears. In the higher order case, the function space may be written as the direct sum

$$\mathcal{P}_K = [P_{q-1}(K)]^3 \oplus S_q,$$

where

$$S_q = \{v \in [\tilde{P}_q(K)]^3 : v \cdot x = 0\}.$$

Here, $\tilde{P}_q(K)$ is the space of homogeneous polynomials of degree q on K. An alternate characterization of $\mathcal{P}_K$ is that it is the space of polynomials of degree $q+1$ on which the qth power of the elastic stress tensor vanishes. The dimension of $\mathcal{P}_q$ is exactly

$$n_K = \frac{q(q+2)(q+3)}{2}.$$

▶ Author note: *What is the qth power of the elastic stress tensor?*

▶ Author note: *What is the dimension on triangles?*

The degrees of freedom are chosen to ensure tangential continuity between elements and thus a well-defined global curl. In the lowest order case, the six degrees of freedom are the average value of the tangential component along each edge of the tetrahedron, hence the term "edge elements". In the more general case, the degrees of freedom are the $q-1$ tangential moments along each edge, moments of the tangential components against $(P_{q-2})^2$ on each face, and moments against $(P_{q-3})^3$ in the interior of the tetrahedron.

For tetrahedra, there also exists another family of elements known as Nedelec elements of the second kind, appearing in (Nédélec, 1986). These have a simpler function space at the expense of more complicated degrees of freedom. The second kind space of order q is simply vectors of polynomials of degree q. The degrees of freedom are integral moments of degree q along each edge together with integral moments against lower-order first-kind bases on the faces and interior.
▶ Author note: *Note different numbering compared to RT, starting at 1, not zero.*

4.9.2 Historical notes

Nédélec's original paper (Nédélec, 1980) provided rectangular and simplicial elements for $H(\mathrm{div})$ and $H(\mathrm{curl})$ based on incomplete function spaces. This built on earlier two-dimensional work for Maxwell's equations (Adam et al., 1980) and extended the work of Raviart and Thomas for $H(\mathrm{div})$ to three dimensions. The second kind elements, appearing in (Nédélec, 1986), extend the Brezzi–Douglas–Marini triangle (Brezzi et al., 1985a) to three dimensions and curl-conforming spaces. We summarize the relation between the Nedelec elements of first and second kind with the Raviart–Thomas and Brezzi–Douglas–Marini elements in Table 4.1.

In many ways, Nédélec's work anticipates the recently introduced finite element exterior calculus (Arnold et al., 2006a), where the first kind spaces appear as $\mathcal{P}_q^-\Lambda^k$ spaces and the second kind as $\mathcal{P}_q\Lambda^k$. Moreover, the use of a differential operator (the elastic strain) in (Nédélec, 1980) to characterize the function space foreshadows the use of differential complexes (Arnold et al., 2006b).
▶ Author note: *Should we change the numbering of the Nedelec elements and Raviart–Thomas elements to start at $q = 1$?*

Simplex	$H(\mathrm{div})$		$H(\mathrm{curl})$	
$K \subset \mathbb{R}^2$	RT_{q-1}	$\mathcal{P}_q^-\Lambda^1(K)$	$\mathrm{NED}_{q-1}(\mathrm{curl})$	—
	BDM_q	$\mathcal{P}_q\Lambda^1(K)$		
$K \subset \mathbb{R}^3$	$\mathrm{RT}_{q-1} = \mathrm{NED}^1_{q-1}(\mathrm{div})$	$\mathcal{P}_q^-\Lambda^2(K)$	$\mathrm{NED}^1_{q-1}(\mathrm{curl})$	$\mathcal{P}_q^-\Lambda^1(K)$
	$\mathrm{BDM}_q = \mathrm{NED}^2_q(\mathrm{div})$	$\mathcal{P}_q\Lambda^2(K)$	$\mathrm{NED}^2_q(\mathrm{curl})$	$\mathcal{P}_q\Lambda^1(K)$

Table 4.1: Nedelec elements of the first and second kind and their relation to the Raviart–Thomas and Brezzi–Douglas–Marini elements as well as to the notation of finite element exterior calculus.

4.10 The PEERS Element

4.10.1 Definition

The PEERS element (Arnold et al., 1984) provides a stable tensor space for discretizing stress in two-dimensional mixed elasticity problems. The stress tensor σ is represented as a 2×2 matrix, each row of which is discretized with a vector-valued finite element. Normally, one expects the stress tensor to be symmetric, although the PEERS element works with a variational formulation that enforces this condition weakly.

The PEERS element is based on the Raviart–Thomas element described in Section 4.11. If $\mathrm{RT}_0(K)$ is the lowest-order Raviart-Thomas function space on a triangle K and b_K is the cubic bubble function that vanishes on ∂K, then the function space for the PEERS element is given by

$$\mathcal{P}_K = \left[\mathrm{RT}_0(K) \oplus \mathrm{span}\{\mathrm{curl}(b_K)\}\right]^2 .$$

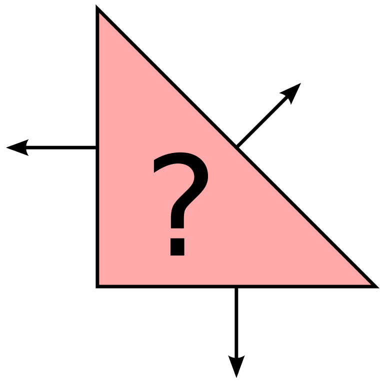


Figure 4.11: The PEERS triangle. One vector-valued component is shown.

▶ Author note*: Which degrees of freedom in the interior? The curl?*

▶ Author note*: Is this really an element? We could also introduce other mixed elements like Taylor–Hood. But perhaps it's suitable to include it since it is not a trivial combination of existing elements (the extra curl part).*

4.10.2 Historical notes

Discretizing the mixed form of planar elasticity is quite a difficult task. Polynomial spaces of symmetric tensors providing inf-sup stability are quite rare, only appearing in the last decade (Arnold and Winther, 2002). A common technique is to relax the symmetry requirement of the tensor, imposing it weakly in a variational formulation. This extended variational form requires the introduction of a new field discretizing the assymetric portion of the stress tensor. When the PEERS element is used for the stress, the displacement is discretized in the space of piecewise constants, and the asymmetric part is discretized in the standard space of continuous piecewise linear elements.

The PEERS element was introduced in (Arnold et al., 1984), and some practical details, including postprocessing and hybridization strategies, are discussed in (Arnold and Brezzi, 1985).

4.11 The Raviart–Thomas Element

4.11.1 Definition

The Raviart–Thomas element, like the Brezzi–Douglas–Marini and Brezzi–Douglas–Fortin–Marini elements, is an $H(\mathrm{div})$-conforming element. The space of order q is constructed to be the smallest polynomial space such that the divergence maps $\mathrm{RT}_q(K)$ onto $P_q(K)$. The function space $\mathcal{P}_K$ is given by

$$\mathcal{P}_K = P_{q-1}(K) + xP_{q-1}(K).$$

The lowest order Raviart–Thomas space thus consists of vector-valued functions of the form

$$v(x) = \alpha + \beta x,$$

where α is a vector-valued constant and β is a scalar constant.

On triangles, the degrees of freedom are the moments of the normal component up to degree q, or, alternatively, the normal component at $q+1$ points per edge. For higher order spaces, these degrees of freedom are supplemented with integrals against a basis for $[P_{q-1}(K)]^2$.

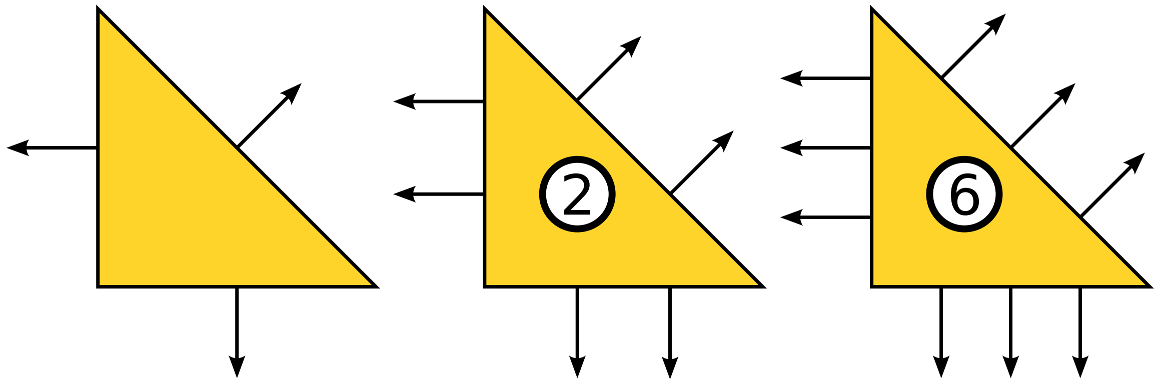


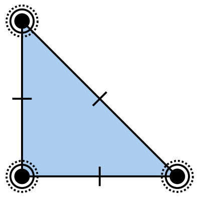
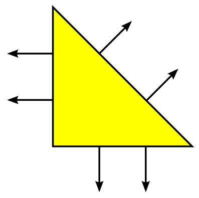
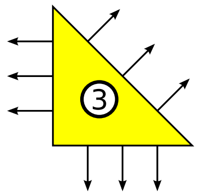
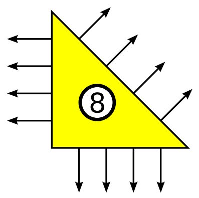
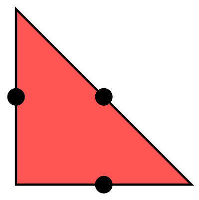
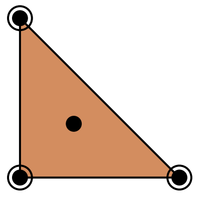
Figure 4.12: The zeroth order, linear and quadratic Raviart–Thomas triangles.

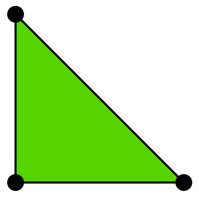
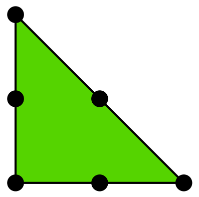
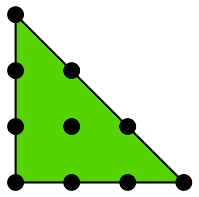
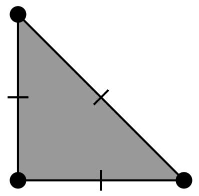
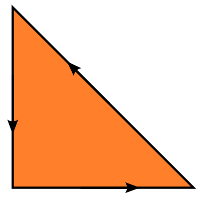
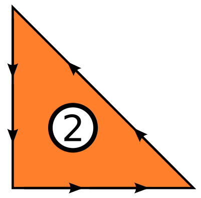
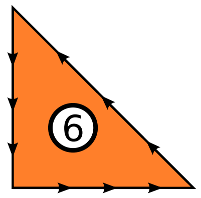
4.11.2 Historical notes

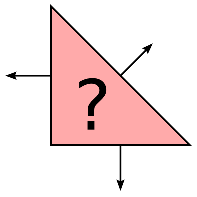
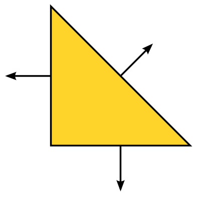
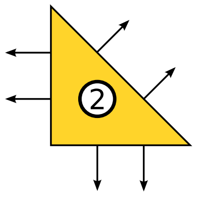
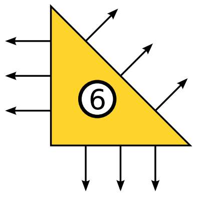
The Raviart–Thomas element was introduced in (Raviart and Thomas, 1977) in the late 1970's, the first element to discretize the mixed form of second order elliptic equations. Shortly thereafter, it was extended to tetrahedra and boxes by Nédélec (Nédélec, 1980) and so is sometimes referred to as the Raviart–Thomas–Nédélec element or a first kind $H(\mathrm{div})$ element.

On rectangles and boxes, there is a natural relation between the lowest order Raviart–Thomas element and cell-centered finite differences. This was explored in (Russell and Wheeler, 1983), where a special quadrature rule was used to diagonalize the mass matrix and eliminate the flux unknowns. Similar techniques are known for triangles (Arbogast et al., 1998), although the stencils are more complicated.

4.12 Summary

Notation	Element family	$\mathcal{L}_K$	$\dim \mathcal{P}_K$	References
ARG_5	Quintic Argyris		21	
BDM_1	Brezzi–Douglas–Marini		6	
BDM_2	Brezzi–Douglas–Marini		12	
BDM_3	Brezzi–Douglas–Marini		20	
CR_1	Crouzeix–Raviart		3	
HERM_q	Hermite		10	

P_1	**Lagrange**		3	
P_2	**Lagrange**		6	
P_3	**Lagrange**		10	
MOR_1	**Morley**		6	
NED_1	**Nédélec**		3	
NED_2	**Nédélec**		8	
NED_3	**Nédélec**		15	

PEERS	**PEERS**		?	
RT_0	**Raviart–Thomas**		3	
RT_0	**Raviart–Thomas**		8	
RT_0	**Raviart–Thomas**		15	

► Author note*: Add references to table.*

► Author note*: Indicate which elements are supported by FIAT and SyFi.*

► Author note*: Include formula for space dimension as function of* q *for all elements.*

CHAPTER 5

Constructing General Reference Finite Elements

By Robert C. Kirby and Kent-Andre Mardal

Chapter ref: **[kirby-1]**

▸ Editor note: *Reuse color figures from chapter [kirby-7] for RT elements etc.*

5.1 Introduction

The finite element literature contains a huge collection of approximating spaces and degrees of freedom, many of which are surveyed in Chapter **??**, Some applications, such as Cahn-Hilliard and other fourth-order problems, can benefit from very smooth finite element bases. While porous media flow requires vector fields discretized by piecewise polynomial functions with normal components continuous across cell boundaries. Many problems in electromagnetism call for the tangentially continuous vector fields obtained by using Nedelec elements (**??**). Many elements are carefully designed to satisfy the *inf-sup* condition (**??**), originally developed to explain stability of discretizations of incompressible flow problems. Additionally, some problems call for low-order discretizations, while others are effectively solved with high-order polynomials.

While the automatic generation of computer programs for finite element methods requires one to confront the panoply of finite element families found in the literature, it also provides a pathway for wider employment of Raviart-Thomas, Nedelec, and other difficult-to-program elements. Ideally, one would like to describe the diverse finite element spaces at an abstract level, whence a computer code discerns how to evaluate and differentiate their basis functions. Such goals are in large part accomplished by the FIAT and SyFi projects, whose implementations are described in later chapters.

Projects like FIAT and SyFi may ultimately remain mysterious to the end user of a finite element system, as interactions with finite element bases are typically mediated through tools that construct the global finite element operators. The end user will typically be satisfied if two conditiones are met. First, a finite element system should support the common elements used in the application area of interest. Second, it should provide flexibility with respect to order of approximation.

It is entirely possible to satisfy many users by *a priori* enumerating a list of finite elements and implement only those. At certain times, this would even seem ideal. For example, after the rash of research that led to elements such as the Raviart-Thomas-Nedelec and Brezzi-Douglas-Marini families, development of new families slowed considerably. Then, more recent work of lead forth by Arnold, Falk, and Winther in the context of exterior calculus has not only led to improved understanding of existing element families, but has also brought a new wave of elements with improved proprerties. A generative system for finite element bases can far more readily assimilate these and future developments. Automation also provides generality with respect to the order of approximation that standard libraries might not otherwise provide. Finally, the end-user might even easilly define his own new element and test its numerical properties before analyzing it mathematically.

In the present chapter, we describe the mathematical formulation underlying such projects as FIAT (**??**), SyFi (Alnæs and Mardal, 2009, **?**) and Exterior (**?**). This formulation starts from definitions of finite elements as given classically by Ciarlet (**?**). It then uses basic linear algebra to construct the appropriate nodal basis for an abstract finite element in terms of polynomials that are easy to implement and well-behaved in floating point arithmetic. We focus on constructing nodal bases for a single, fixed reference element. As we will see in Chapter **??**, form compilers such as `ffc` (Logg, 2007) and `sfc` (**?**) will work in terms of this single reference element.

Other approaches exist in the literature, such as the hierarchical bases studied by Szabo and Babuska (**?**) and extended to $H(\mathrm{curl})$ and $H(\mathrm{div})$ spaces in work such as (**?**). These can provide greater flexibility for refining the mesh and polynomial degree in finite element methods, but they also require more care during assembly and are typically constructed on a case-by-case basis for each element family. When they are available, they may be cheaper to construct than using the technique studied here, but this present technique is easier to apply to an "arbitrary" finite element and so is considered in the context of automatic software.

5.2 Preliminaries

Both FIAT and SyFi work a slightly modified version of the abstract definition of a finite element introduced by Ciarlet (**?**).

Definition 5.1 (A Finite Element) *A finite element is a triple* (K, P, N) *with*

1. $K \subset \mathbb{R}^d$ *a bounded domain with piecewise smooth boundary.*
2. *A finite-dimensional space* P *of* $BC^m(K, V)$*, where* V *is some normed vector space and* BC^m *is the space of* m*-times bounded and continuosly differentiable functions from* K *into* V.
3. *A dual basis for* P*, written* $N = \{L_i\}_{i=1}^{\dim P}$*. These are bounded linear functionals in* $(BC^m(K, V))'$ *and frequently called the* nodes *or* degrees of freedom.

In this definition, the term “finite element” refers not only to a particular cell in a mesh, but also to the associated function space and degrees of freedom. Typically, the domain K is some simple polygonal or polyhedral shape and the function space P consists of polynomials.

Given a finite element, a concrete basis, often called the nodal basis, for this element can be computed by using the following defintion.

Definition 5.2 *The* nodal basis *for a finite element* (K, P, N) *be a finite element is the set of functions* $\{\psi_i\}_{i=1}^{\dim P}$ *such that for all* $1 \leq i, j \leq \dim P$*,*

$$L_i(\psi_j) = \delta_{i,j}. \tag{5.1}$$

The main issue at hand in this chapter is the *construction* of this nodal basis. For any given finite element, one may construct the nodal basis explicitly with elementary algebra. However, this becomes tedious as we consider many different familes of elements and want arbitrary order instances of each family. Hence, we present a linear algebraic paradigm here that undergirds computer programs for automating the construction of nodal bases.

In addition to the construction of the nodal base we need to keep in mind that finite elements are patched together to form a piecewise polynomial field over a mesh. The fitness (or stability) of a particular finite element method for a particular problem relies on the continuity requirements of the problem. The degrees of freedom of a particular element are often choosen such that these continuity requirements are fulfilled.

Hence, in addition to computing the nodal basis, the mathematical structure developed here simplifies software for the following tasks:

1. Evaluate the basis function and their derivatives at points.

2. Associtate the basis function (or degree of freedom) with topological facets of K such as its vertices, edges and its placement on the edges.

3. Associtate the basis function with additional metadata such as a sign or a mapping that should be used together with the evaluation of the basis functions or its derivatives.

4. Provide rules for the degrees of freedom applied to arbitrary input functions determined at run-time.

The first of these is relatively simple in the framework of symbolic computation (SyFi), but they require more care if an implementation uses numerical arithmetic (FIAT). The middle two encode the necessary information for a client code to transform the reference element and assemble global degrees of freedom when the finite element is either less or more than C^0 continuous. The final task may take the form of a point at which data is evaluated or differentiated or more generally as the form of a sum over points and weights, much like a quadrature rule.

A common practice, employed throuought the FEniCS software and in many other finite element codes, is to map the nodal basis functions from this reference element to each cell in a mesh. Sometimes, this is as simple as an affine change of coordinates; in other cases it is more complicated. For completeness, we briefly describe the basics of creating the global finite elements in terms of a mapped reference element. Let therefore T be a polygon and $\hat{T}$ the corresponding reference polygon. Between the coordinates $\mathrm{x} \in T$ and $\mathrm{x}i \in \hat{T}$ we use the mapping

$$\mathbf{x} = \mathbf{G}(\mathbf{x}i) + \mathbf{x}_0, \tag{5.2}$$

and define the Jacobian determinant of this mapping as

$$J(\mathbf{x}) = \left| \frac{\partial \mathbf{G}(\mathbf{x}i)}{\partial \mathbf{x}i} \right|. \tag{5.3}$$

The basis functions are defined in terms of the basis function on the reference element as

$$N_j(\mathbf{x}) = \hat{N}_j(\mathbf{x}i), \tag{5.4}$$

where $\hat{N}_j$ is basis function number j on the reference element. The integral can then be performed on the reference polygon,

$$\int_T f(\mathbf{x})\, d\mathbf{x} = \int_{\hat{T}} f(\mathbf{x}i)\, J d\mathbf{x}i, \tag{5.5}$$

and the spatial derivatives are defined by the derivatives on the reference element and the geometry mapping simply by using the chain rule,

$$\frac{\partial N}{\partial x_i} = \frac{\partial N}{\partial \xi_j} \frac{\partial \xi_j}{\partial x_i}. \tag{5.6}$$

The above definition of a global element is commonly called *isoparametric* and is common for approximations in H^1. For approximations in $H(div)$ or $H(curl)$ it is neccessary to modify (5.4) with the Piola mapping. Furthermore, some elements like the Rannacher-Turek element (**??**) has far better properties when defined globally compared its analogous definition in terms of a reference element.

5.3 Mathematical Framework

5.3.1 Change of basis

► Editor note: *Coordinate with notation in Chapter 3 where B is used for the Vandermonde matrix. Perhaps we could use $\mathcal{V}$? Also use ℓ for the functionals and α for the expansion coefficients.*

The fundamental idea in constructing nodal basis is from elementary linear algebra: one constructs the desired (nodal) basis as a linear combination of a basis one has "in hand". We will start with some basis $\{\phi\}_{i=1}^{\dim P} \subset P$. From this, we construct each nodal basis function

$$\psi_j = A_{jk}\phi_k, \tag{5.7}$$

where summation is implied over the repeated index k. The task is to compute the matrix A. Each fixed ψ_j must satisfy

$$L_i(\psi_j) = \delta_{i,j}, \tag{5.8}$$

and using the above expansion for ψ_j, we obtain

$$\delta_{i,j} = L_i(A_{jk}\phi_k) = A_{jk}L_i(\phi_k). \tag{5.9}$$

So, for a fixed j, we have a system of $\dim P$ equations

$$V_{ik}A_{jk} = e^j, \tag{5.10}$$

where

$$V_{ik} = L_i(\phi_k) \tag{5.11}$$

is a kind of generalized Vandermonde matrix and e^j is the canonical basis vector. Of course, (5.10) can be used to write a matrix equation for A as a linear system with $\dim P$ right hand sides and obtain

$$VA^t = I. \tag{5.12}$$

In practice, this, supposes that one has an implementation of the original basis for which the actions of the nodes (evaluation, differentiation, and integration) may be readily computed.

This discussion may be summarized as a proposition.

Proposition 5.3.1 *Define V and A as above. Then*

$$V = A^{-t}. \tag{5.13}$$

5.3.2 Polynomial spaces

In Definition 5.1 we defined the finite element in terms of a finite dimensional function space P. Although it is not strictly neccessary, the functions used in finite elements are typically polynomials. While our general strategy will in principle accomodate non-polynomial bases, we only deal with polynomials in this chapter. A common space is $\mathbb{P}_n^d$, the space of polynomials of degree n in $\mathbb{R}^d$. There are many different ways to represent $\mathbb{P}_n^d$. We will discuss the power basis, orthogonal bases, and the Bernstein basis. Each of these bases has explicit representations and well-known evaluation algorithms. In addition to $\mathbb{P}_n^d$ we will also for some elements need $\mathbb{H}_n^d$, the space of homogenous polynomials of degree n in d variables.

Typically, the developed techniques here are used on simplices, where polynomials do not have a nice tensor-product structure. Some rectangular elements like the Brezzi-Douglas-Marini family (**?**), however, are not based on tensor-product polynomial spaces, and the techniques described in this chapther apply in that case as well. SyFi has explicit support for rectangular domains, but FIAT does not.

Power basis

On a line segment, $\mathbb{P}_n^1 = \mathbb{P}_n$ the monomial, or power basis is $\{x^i\}_{i=0}^n$, so that any $v \in P_n$ is written as

$$v = a_0 + a_1 x + \ldots a_n x^n = \sum_{i=0}^{n} a_i x^i. \tag{5.14}$$

In 2D on triangles, $\mathbb{P}_n$ is spanned by functions on the form $\{x^i y^j\}_{i,j=0}^{i+j\leq n}$, with a similar definition in three dimensions.

This basis is quite easy to evaluate, differentiate, and integrate but is very ill-conditioned in numerical calculations.

Legendre basis

A popular polynomial basis for polygons that are either intervals, rectangles or boxes are the Legendre polynomials. This polynomial basis is also usable to represent polynomials of high degree. The basis is defined on the interval $[-1, 1]$, as

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1), \quad k = 0, 1, \ldots,$$

A nice feature with these polynomials is that they are orthogonal with respect to the L_2 inner product, i.e.,

$$\int_{-1}^{1} L_k(x) L_l(x)\,dx = \left\{ \begin{array}{c} \frac{2}{2k+1},\ k = l, \\ 0,\ \ k \neq l, \end{array} \right.$$

The Legendre polynomials are extended to rectangular domains in 2D and 3D simply by taking the tensor product,

$$L_{k,l,m}(x,y,z) = L_k(x)L_l(y)L_m(z).$$

and $\mathbb{P}^n$ is defined by functions on the form (in 3D),

$$v = \sum_{k,l,m=0}^{k,l,m<=n} a_{k,l,m}L_{k,l,m}.$$

Dubiner basis

Orthogonal polynomials in simplicial domains are also known, although they lack some of the rotational symmetry of the Legendre polynomials. The Dubiner basis, frequently used in simplicial spectral elements (**?**), is known under many names in the literature. It is an L^2-orthogonal basis that can be constructed by mapping particular tensor products of Jacobi polynomials on a square by a singular coordinate change to a fixed triangle. Let $P_n^{\alpha,\beta}$ denote the n^{th} Jacobi polynomial with weights α, β. Then, define the new coordinates

$$\begin{aligned} \eta_1 &= 2\left(\frac{1+x}{1-y}\right) - 1 \\ \eta_2 &= y, \end{aligned} \tag{5.15}$$

which map the square $[-1,1]^2$ to the triangle with vertices $(-1,-1), (-1,1), (1,-1)$ as shown in Figure **??**. This is the natural domain for defining the Dubiner polynomials, but they may easily be mapped to other domains like the the triangle with vertices $(0,0), (0,1), (1,0)$ by an affine mapping. Then, one defines

$$\phi^{p,q}(x,y) = P_p^{0,0}(\eta_1)\left(\frac{1-\eta_2}{2}\right)^p P_q^{2p+1,0}(\eta_2). \tag{5.16}$$

Though it is not obvious from the definition, $\phi^{p,q}(x,y)$ is a polynomial in x and y of degree $p+q$. Moreover, for $(p,q) \neq (i,j)$, $\phi^{p,q}$ is L^2-orthogonal to $\phi^{i,j}$.

While this basis is more complicated than the power basis, it is very well-conditioned for numerical calculations even with high degree polynomials. The polynomials can also be ordered hierarchically so that $\{\phi_i\}_{i=1}^{\dim P_k}$ forms a basis for polynomials of degree k for each $k > 0$. As a possible disadvantage, the basis lacks rotational symmetry that can be found in other bases.

Bernstein basis

The Bernstein basis is another well-conditioned basis that can be used in generating finite element bases. In 1D, the basis functions take the form,

$$B_{i,n} = \binom{i}{n} x^i(1-x)^{n-i}, \quad i = 0, \ldots, n,$$

and then P_n is spanned by $\{B_{i,n}\}_{i=0}^{n}$ And with this basis, $\mathbb{P}_n$ can be spanned by functions on the form,

The terms x and $1-x$ appearing in $B_{i,n}$ are the barycentric coordinates for $[0,1]$, an observation that makes it easy to extend the basis to simplices in higher dimensions.

Let b_1, b_2, and b_3 be the barycentric coordinates for the triangle shown in Figure **??**, i.e., $b_1 = x$, $b_2 = y$, and $b_3 = 1 - x - y$. Then the basis is of the form,

$$B_{i,j,k,n} = \frac{n!}{i!j!k!} b_1^i b_2^j b_3^k, \quad for\ i + j + k = n.$$

and a basis for $\mathbb{P}_n$ is simply.

$$\{B_{i,j,k,n}\}_{i,j,k\geq 0}^{i+j+k=n}.$$

The Bernstein polynomials on the tetrahedron are completely analogous (**?**).

These polynomials, though less common in the finite element community, are well-known in graphics and splines. They have a great deal of rotational symmetry and are totally nonnegative and so give positive mass matrices, though they are not hierarchical.

Homogeneous Polynomials

Another set of polynomials which sometimes are useful are the set of homogeneous polynomials $\mathbb{H}^n$. These are polynomials where all terms have the same degree. $\mathbb{H}^n$ is in 2D spanned by polynomials on the form:

$$v = \sum_{i,j,\ i+j=n} a_{i,j,k} x^i y^j$$

with a similar definition in nD.

Vector or Tensor valued Polynomials

It is straightforward to generalize the scalar valued polynomials discussed earlier to vector or tensor valued polynomials. Let $\{e_i\}$ be canonical basis in $\mathbb{R}^d$. Then a basis for the vector valued polynomials is

$$P_{ij} = P_j \mathbf{e}_i,$$

with a similar definition extending the basis to tensors.

5.4 Examples of Elements

We include some standard finite elements to illustrate the concepts and motivate the need for different finite elements. Notice that the different continuity of the elements result in different approximation properties. We refer the reader to Chapter (**?**) for a more thorough review of elements.

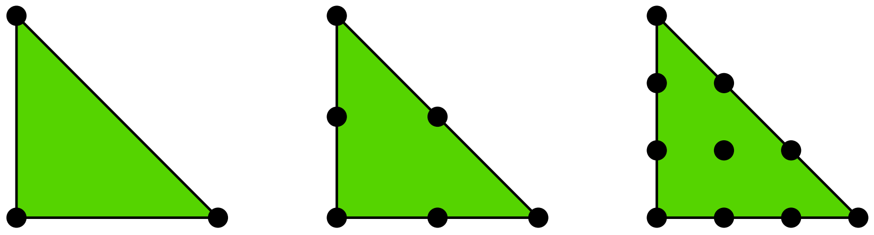


Figure 5.1: Lagrangian elements of order 1, 2, and 3.

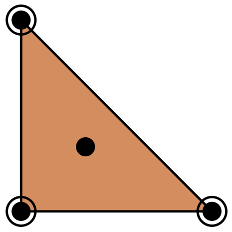


Figure 5.2: Hermite elements of order 3.

Example 5.1 The Lagrange Element

The Lagrangian element shown in Figure 5.1 is the most common element, where the black dots mean point evaluation. The first order element is shown in the leftmost triangle, it consists of three degrees of freedom in each of the vertices. The cooresponding basis functions are x, y, and $1-x-y$. The second order element is shown in middle triangle, it has six degrees of freedom, three at the vertices and three at the edges. Finally, we have the third order Lagrangian element in the rightmost triangle, with ten degrees of freedom.

The Lagrangian element produce piecewise continuous polynomials and they are therefore well suited for approximation in H^1. In general the number of degress of freedom $\mathbb{P}_n$ in 2D is $(n+2)(n+1)/2$, which is the same as the number of degrees of freedom in the Lagrange element. In fact, on a simplex in any dimension d the degrees of freedom of the Lagrange element of order n is the same as $\mathbb{P}_n^d$.

Example 5.2 The Hermite Element

In Figure 5.2 we show the Hermite element. The black dots mean point evaluation, while the white circles mean evaluation of derivatives in both x and y direction. Hence, the degrees of freedom for this element is three point evaluations at the vertices + six derivatives in the vertices + one internal point evaluation, which in total is ten degrees of freedom, which is the same number of degrees of

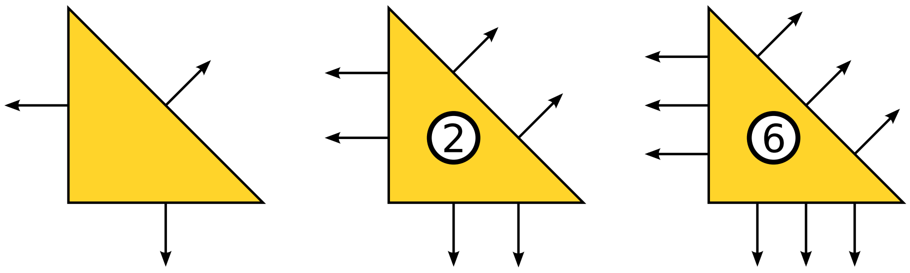


Figure 5.3: Triangular Raviart–Thomas elements of order 0, 1, and 2.

freedom as in $\mathbb{P}_3^2$. The advantage of the Hermite element is that it has continuous derivatives at the vertices (it will however not neccessarily result in a H^2 conforming approximation).

Example 5.3 The Raviart-Thomas Element

In Figure 5.3 we illustrate the Raviart-Thomas element. In contrast to the previous elements, this element has a vector-valued function space. The arrows represent normal vectors while the double dots indicate pointwise evaluation in both the $x-$ and $y-$ direction. Starting at $n = 0$, the dimension of RT_n is $(n+1)(n+3)$. The Raviart-Thomas element is typically used for approximations in $H(\text{div})$.

Remark 5.4.1 *Earlier we saw common bases for $\mathbb{P}_d^n$, but elements like the Raviart-Thomas element described above use function spaces other than $\mathbb{P}_d^n$ or vectors or tensors thereof. To fix this, we must either construct a basis for the appropriate polynomial space or work with a different element that includes full vectors of $\mathbb{P}_d^n$. In the case of $H(\text{div})$ elements, this corresponds to using the Brezzi-Douglas-Fortin-Marini elements.*

5.4.1 Bases for other polynomial spaces

The basis presented above are suitable for constructing many finite elements, but as we have just seen, they do not work in all cases. The Raviart-Thomas function space,

$$\left(\mathbb{P}_n^2\right)^2 \oplus \begin{pmatrix} x \\ y \end{pmatrix} \mathbb{H}_n^2,$$

requires a basis for vectors of polynomials to be supplemented with an extra $\dim \mathbb{H}_n^2 = \dim \mathbb{P}_n^2 - \dim \mathbb{P}_{n-1}^2 = n$ functions. In fact, any n polynomials in $\mathbb{P}_n^2 \backslash \mathbb{P}_{n-1}^2$ will work, not just homogeneous polynomials, although the sum above will no longer be direct.

While the Raviart-Thomas element requires supplementing a standard basis with extra functions, the Brezzi-Douglas-Fortin-Marini triangle uses the function space

$$\left\{u \in (\mathbb{P}^2_n(K))^2 : u \cdot n \in \mathbb{P}^1_{n-1}(\gamma), \gamma \in \mathcal{E}(K)\right\}.$$

Obtaining a basis for this space is somewhat more subtle. FIAT and SyFi have developed different but mathematically equivalent solutions. Both rely on recognizing that the required function space sits inside $\left(\mathbb{P}^2_n\right)^2$ and can be characterized by certain functionals that vanish on this larger space.

Three such functionals describe the basis for the BDFM triangle. If μ^γ_n is the Legendre polynomial of order n on an edge γ, then the functional

$$\ell_\gamma(u) = \int_\gamma (u \cdot n)\mu^\gamma_n$$

acting on $\left(\mathbb{P}^2_n\right)^2$ must vanish on the BDFM space, for the n^{th} degree Legendre polynomial is orthogonal to all polynomials of lower degree.

Now, we define the matrix

$$V = \begin{pmatrix} V^1 \\ V^2 \end{pmatrix}. \tag{5.17}$$

Here, $V^1 \in \mathbb{R}^{2\dim \mathbb{P}^2_n - 3, 2\dim \mathbb{P}^2_n}$ and $V^2 \in \mathbb{R}^{3, 2\dim \mathbb{P}^2_n}$ are defined by

$$V^1_{ij} = L_i(\phi_j),$$
$$V^2_{ij} = \ell_i(\phi_j),$$

where $\{\phi_j\}_{j=1}^{2\dim \mathbb{P}^2_n}$ is a basis for $(\mathbb{P}^2_n)^2$.

Consider now the matrix equation

$$VA^t = I_{2\dim P_n, 2\dim P_n - 3}, \tag{5.18}$$

where $I_{m,n}$ denotes the $m \times n$ identity matrix with $I_{i,j} = \delta_{i,j}$. As before, the columns of A still contain the expansion coefficients of the nodal basis functions ψ_i in terms of $\{\phi_j\}$. Moreover, $V_2 A = 0$, which imples that the nodal basis functions are in fact in the BDFM space.

More generally, we can think of our finite element space P being embedded in some larger space $\bar{P}$ for which there is a readily usable basis. If we may characterize P by

$$P = \cap_{i=1}^{\dim \bar{P} - \dim P} \ell_i,$$

where $\ell_i : \bar{P} \to \mathbb{R}$ are linear functionals, then we may apply this technique. In this case, $V_1 \in \mathbb{R}^{\dim P, \dim \bar{P}}$ and $V_2 \in \mathbb{R}^{\dim \bar{P} - \dim P, \dim \bar{P}}$. This scenario, though somewhat abstract, is applicable not only to BDFM, but also to the Nédélec (**?**), Arnold-Winther (Arnold and Winther, 2002), Mardal-Tai-Winther (**?**), Tai-Winther (**?**), and Bell (**?**) element families.

Again, we summarize the discussion as a proposition.

Proposition 5.4.1 *Let (K, P, N) be a finite element with $P \subset \bar{P}$. Let $\{\phi_i\}_{i=1}^{\dim \bar{P}}$ be a basis for $\bar{P}$. Suppose that there exist functionals $\{\ell_i\}_{i=1}^{\dim \bar{P}-\dim P}$ on $\bar{P}$ such that*

$$P = \cap_{i=1}^{\dim \bar{P}-\dim P} \text{null}(\ell_i).$$

Define the matrix A as in (5.18). Then, the nodal basis for P may be expressed as

$$\psi_i = A_{ij}\phi_j,$$

where $1 \leq i \leq \dim P$ and $1 \leq j \leq \dim \bar{P}$.

5.5 Operations on the Polynomial spaces

Here, we show various important operations may be cast in terms of linear algebra, supposing that they may be done on original basis $\{\phi_i\}_{i=1}^{\dim P}$.

5.5.1 Evaluation

In order to evaluate the nodal basis $\{\psi_i\}_{i=1}^{\dim P}$ at a given point $x \in K$, one simply computes the vector

$$\Phi_i = \phi_i(x)$$

followed by the product

$$\psi_i(x) \equiv \Psi_i = A_{ij}\Phi_j.$$

Generally, the nodal basis functions are required at an array of points $\{x_j\}_{j=1}^m \subset K$. For performance reasons, performing matrix-matrix products may be advantageous. So, define $\Phi_{ij} = \Phi_i(x_j)$ and $\Psi_{ij} = \Psi_i(x_j)$. Then all of the nodal basis functions may be evaluated by the product

$$\Psi_{ij} = A_{ik}\Phi_{kj}.$$

5.5.2 Differentiation

Differentiation is more complicated, and also presents more options. We want to handle the possibility of higher order derivatives. Let $\alpha = (\alpha_1, \alpha_2, \ldots \alpha_d)$ be a multiindex so that

$$D^\alpha \equiv \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_d^{\alpha_d}},$$

where $|\alpha| = \sum_{i=1}^d \alpha_i$.

We want to compute the array

$$\Psi_i^\alpha = D^\alpha \psi_i(x)$$

for some $x \in K$.

One may differentiate the original basis functions $\{\phi_i\}$ to produce an array

$$\Phi_i^\alpha = D^\alpha \psi_i(x),$$

whence

$$\Psi_i^\alpha = A_{ij}\Phi_i^\alpha.$$

This presupposes that one may conveniently compute all derivatives of the $\{\phi_i\}$. This is typically true in symbolic computation or when using the power basis. Alternatively, automatic differentiation (**?**) could be used in the power or Bernstein basis. The Dubiner basis, as typically formulated, contains a coordinate singularity that prevents automatic differentiation from working at the top vertex. Recent work (**?**) has reformulated recurrence relations to allow for this possibility.

If one prefers (or is required by the particular starting basis), one may also compute matrices that encode first derivatives acting on the $\{\phi_i\}$ and construct higher derivatives than these. For each coordinate direction x_k, a matrix D^k is constructed so that

$$\frac{\partial \phi_i}{\partial x_i} = D^k_{ij}\phi_j.$$

How to do this depends on which bases are chosen. For particular details on the Dubiner basis, see (**?**). Then, Ψ^α may be computed by

$$\Psi_i^\alpha = (\mathrm{D}^\alpha A)_{ij}\Phi_j,$$

5.5.3 Integration

Integration of basis functions over K, including products of basis functions and/or their derivatives, may be performed numerically, symbolically, or exactly with some known formula. An important aspect of automation is that it allows general orders of approximation. While this is not difficult in symbolic calculations, a numerical implementation like FIAT must work harder. If Bernstein polynomials are used, we may use the formula

$$\int_K b_1^i b_2^j b_3^k \, dx = \frac{i!j!k!}{(i+j+k+2)!}\frac{|K|}{2}$$

on triangles and a similar formula for lines and tetrahedra to calculate integrals of things expressed in terms of these polynomials exactly. Alternately, if the Dubiner basis is used, orthogonality may be used. In either case, when derivatives occur, it may be as efficient to use numerical quadrature. On rectangular domains, tensor products of Gauss or Gauss-Lobatto quadrature can be used to give efficient quadrature families to any order accuracy. On the simplex, however, optimal quadrature is more difficult to find. Rules based on barycentric symmetry (**?**) may be used up to a certain degree (which is frequently high enough in practice). If one needs to go beyond these rules, it is possible to use the mapping (5.15) to map tensor products of Gauss-Jacobi quadrature rules from the square to the triangle.

5.5.4 Linear functionals

Integration, pointwise evaluation and differentiation all are examples of linear functionals. In each case, the functionals are evaluated by their action on the $\{\phi_i\}$

CHAPTER 6

Finite Element Variational Forms

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-5]**

In Chapter [kirby-7], we introduced the following canonical variational problem: Find $u \in V$ such that

$$a(v, u) = L(v) \quad \forall v \in \hat{V}, \tag{6.1}$$

where $\hat{V}$ is a given test space and V is a given trial space. The bilinear form

$$a : \hat{V} \times V \to \mathbb{R}$$

maps a pair of test and trial functions to a real number and is linear in both arguments. Similarly, the linear form $L : \hat{V} \to \mathbb{R}$ maps a given test function to a real number. We also considered the discretization of nonlinear variational problems: Find $u \in V$ such that

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

Here, $F : \hat{V} \times V \to \mathbb{R}$ again maps a pair of functions to a real number. The semilinear form F is linear in the test function v but possibly nonlinear in the trial function u. Alternatively, we may consider the mapping

$$L_u \equiv F(\cdot; u) : \hat{V} \to \mathbb{R},$$

and note that L_u is a linear form on $\hat{V}$ for any fixed value of u.

In Chapter [kirby-7], we also considered the estimation of the error in a given functional $\mathcal{M} : V \to \mathbb{R}$. Here, the possibly nonlinear functional $\mathcal{M}$ maps a given function u to a real number $\mathcal{M}(u)$.

In all these examples, the central concept is that of a form that maps a given tuple of functions to a real number. We shall refer to these as *multilinear forms*. Below, we formalize the concept of a multilinear form, discuss the discretization of multilinear forms, and related concepts such as the *action* of a multilinear form.

Much of the FEniCS software is devoted to the formulation of multilinear forms (UFL), the discretization of multilinear forms (FIAT, FFC, SyFi) and the assembly of the corresponding discrete operators (DOLFIN).

6.1 Multilinear Forms

A form is a mapping from the product of a given sequence $\{V_j\}_{j=1}^{\rho}$ of function spaces to a real number,

$$a : V_1 \times V_2 \times \cdots \times V_\rho \to \mathbb{R}.$$

If the form a is linear in each of its arguments, we say that the form is *multilinear*. The number of arguments ρ of the form is the *arity* of the form.

Forms may often be parametrized over one or more *coefficients*. A typical example is the right-hand side L of the canonical variational problem (6.1), which is a linear form parametrized over a given coefficient f. We shall use the notation $a(v; f) \equiv L_f(v) \equiv L(v)$ and refer to the test function v as an *argument* and to the function f as a *coefficient*. In general, we shall refer to forms which are linear in each argument (but possibly nonlinear in its coefficients) as multilinear forms. Such a multilinear form is a mapping from the product of a sequence of argument spaces $\{V_j\}_{j=1}^{\rho}$ and a sequence of coefficient spaces $\{W_j\}_{j=1}^{n}$,

$$\begin{aligned}
&a : V_1 \times V_2 \times \cdots \times V_\rho \, \times \, W_1 \times W_2 \times \cdots \times W_n \to \mathbb{R}, \\
&a \mapsto a(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots, w_n).
\end{aligned}$$

The argument spaces $\{V_j\}_{j=1}^{\rho}$ and coefficient spaces $\{W_j\}_{j=1}^{n}$ may all be the same space but they typically differ, when Dirichlet boundary conditions are imposed on one or more of the spaces, or when the multilinear form arises from the discretization of a mixed problem such as in Section **??**.

In finite element applications, the arity of a form is typically $\rho = 2$, in which case the form is said to be bilinear, or $\rho = 1$, in which case the form is said to be linear. In the special case of $\rho = 0$, we shall refer to the multilinear form as a *functional*. It may sometimes also be of interest to consider forms of higher arity ($\rho > 2$). Below, we give examples of some multilinear forms of different arity.

6.1.1 Examples

Poisson's equation

Consider Poisson's equation with variable conductivity $\kappa = \kappa(x)$,

$$-\nabla \cdot (\kappa \nabla u) = f.$$

Assuming Dirichlet boundary conditions on the boundary $\partial\Omega$ of the domain Ω, the corresponding canonical variational problem is defined in terms of a pair of multilinear forms, $a(v, u) = \int_\Omega \kappa \nabla v \cdot \nabla u \, \mathrm{d}x$ and $L(v) = \int_\Omega v f \, \mathrm{d}x$. Here, a is a bilinear form ($\rho = 2$) and L is a linear form ($\rho = 1$). Both forms have one coefficient ($n = 1$) and the coefficients are κ and f respectively:

$$\begin{aligned} a &= a(v, u; \kappa), \\ L &= L(v; f). \end{aligned}$$

We usually drop the coefficients from the notation and use the short-hand notation $a = a(v, u)$ and $L = L(v)$.

The incompressible Navier–Stokes equations

The incompressible Navier–Stokes equations for the velocity u and pressure p of an incompressible fluid read:

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

where the stress tensor σ is given by $\sigma(u, p) = 2\mu\epsilon(u) - pI$ and ϵ is the symmetric gradient, $\epsilon(u) = \frac{1}{2}(\nabla u + (\nabla u)^\top)$. Consider here the form obtained by integrating the nonlinear term $\nabla u \cdot u$ against a test function v,

$$a(v; u) = \int_\Omega v \cdot \nabla u \cdot u \, \mathrm{d}x.$$

This is a linear form ($\rho = 1$) with one coefficient ($n = 1$). We may linearize $u = \bar{u} + \delta u$ around a fixed velocity $\bar{u}$:

$$a(v; u) = a(v; \bar{u}) + a'(v; \bar{u})\delta u + \mathcal{O}(\delta u^2).$$

The linearized operator a' is here given by

$$a'(v, \delta u; \bar{u}) \equiv a'(v; \bar{u})\delta u = v \cdot \nabla \delta u \cdot \bar{u} + v \cdot \nabla \bar{u} \cdot \delta u.$$

This is a bilinear form ($\rho = 2$) with one coefficient ($n = 1$). We may also consider the *trilinear* form

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

where w may be a given (frozen) value for the convective velocity in a fixed point iteration for the Navier–Stokes equations. Such a trilinear form may be assembled into a rank three tensor and applied to a given vector of expansion coefficients for w to obtain a rank two tensor (a matrix) corresponding to the bilinear form $a(v, u; w)$. This is rarely done in practice due to the cost of assembling the global rank three tensor. However, the corresponding local rank three tensor may be contracted with the local expansion coefficients for w on each local cell to compute the matrix corresponding to $a(v, u; w)$. We return to this issue below in Chapter [logg-4].

Lift and drag

When solving the Navier–Stokes equations, it may be of interest to compute the lift and drag of some object immersed in the fluid. The lift and drag are given by the z- and x-components of the force generated on the object (for a flow in the x-direction):

$$L_{\text{lift}}(; u, p) = \int_\Gamma \sigma(u, p) \cdot \hat{n} \cdot \hat{e}_z \,\mathrm{d}s,$$
$$L_{\text{drag}}(; u, p) = \int_\Gamma \sigma(u, p) \cdot \hat{n} \cdot \hat{e}_x \,\mathrm{d}s.$$

Here, Γ is the boundary of the body, $\hat{n}$ is the outward unit normal of Γ and $\hat{e}_x$, $\hat{e}_z$ are unit vectors in the x- and z-directions respectively. The arity of both forms is $\rho = 0$ and both forms have two coefficients.

6.1.2 Canonical Form

FEniCS automatically handles the representation and evaluation of a large class of multilinear forms, but not all. In particular, FEniCS is currently limited to forms that may be expressed as a sum of integrals over the cells (the domain), the exterior facets (the boundary) and the interior facets of a given mesh. In particular, FEniCS handles forms that may be expressed as the following canonical form:

$$a(v_1, v_2, \dots, v_\rho; w_1, w_2, \dots, w_n) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_f} \int_{\partial\Omega_k} I_k^f \,\mathrm{d}s + \sum_{k=1}^{n_f^0} \int_{\partial\Omega_k^0} I_k^{f,0} \,\mathrm{d}S. \quad (6.2)$$

Here, each Ω_k denotes a union of mesh cells covering a subset of the computational domain Ω. Similarly, each $\partial\Omega_k$ denotes a subset of the facets on the boundary of the mesh and $\partial\Omega_k^0$ denotes a subset of the interior facets of the mesh. The latter is of particular interest for the formulation of discontinuous Galerkin methods that typically involve integrals across cell boundaries (interior facets).

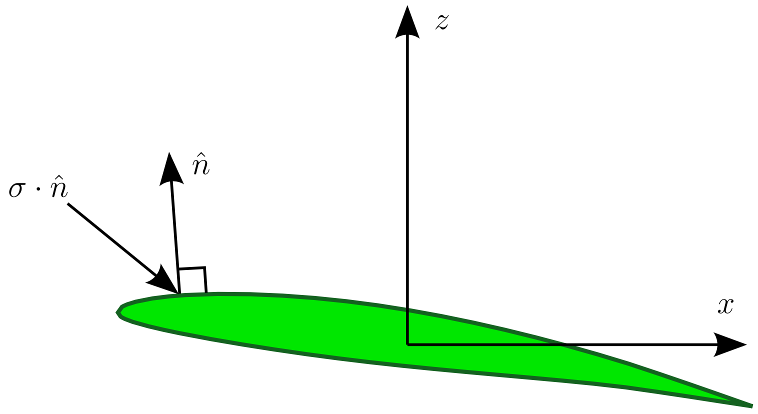


Figure 6.1: The lift and drag of an object, here a NACA 63A409 airfoil, are the integrals of the vertical and horizontal components respectively of the stress $\sigma \cdot \hat{n}$ over the surface Γ of the object. At each point, the product of the stress tensor σ and the outward unit normal vector $\hat{n}$ gives the force per unit area acting on the surface.

One may consider extensions of (6.2) that involve point values or integrals over subsets of cells or facets. Such extensions are currently not supported by FEniCS but may be added in the future.

6.2 Discretizing Multilinear Forms

As we saw in Chapter [kirby-7], one may obtain the finite element approximation $u_h = \sum_{j=1}^{N} U_j \phi_j \approx u$ of the canonical variational problem (6.1) by solving a linear system $AU = b$, where

$$
\begin{aligned}
A_{ij} &= a(\hat{\phi}_i, \phi_j), \quad i, j = 1, 2, \ldots, N, \\
b_i &= L(\hat{\phi}_i), \quad i = 1, 2, \ldots, N.
\end{aligned}
$$

Here, A and b are the discrete operators corresponding to the bilinear and linear forms a and L for the given bases of the test and trial spaces. In general, we may discretize a multilinear form a of arity ρ to obtain a tensor A of rank ρ. The discrete operator A is defined by

$$
A_i = a(\phi^1_{i_1}, \phi^2_{i_2}, \ldots, \phi^\rho_{i_\rho}; w_1, w_2, \ldots, w_n),
$$

where $i = (i_1, i_2, \ldots, i_\rho)$ is a multiindex of length ρ and $\{\phi^j_k\}_{k=1}^{N_j}$ is a basis for $V^j_h \subset V^j$, $j = 1, 2, \ldots, \rho$. The discrete operator is a typically sparse tensor of rank ρ and dimension $N_1 \times N_2 \times \cdots \times N_\rho$.

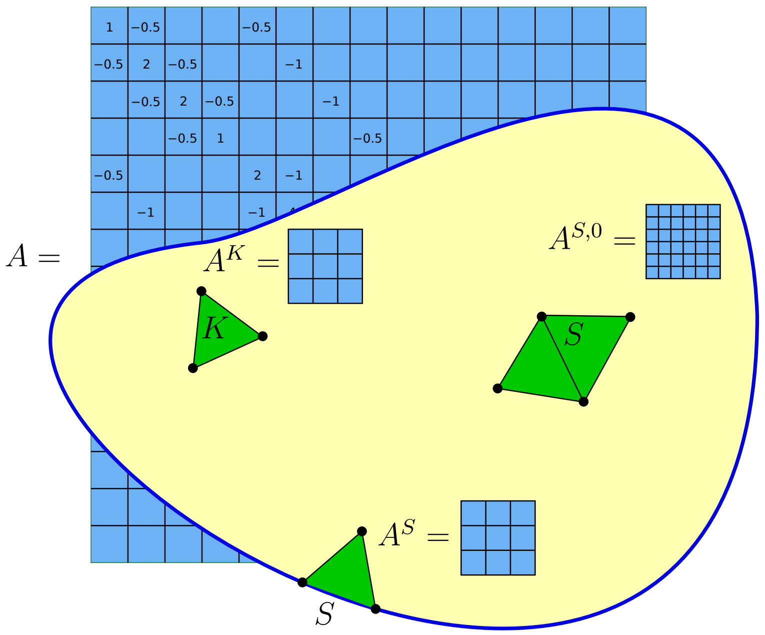


Figure 6.2: The cell tensor A^K, exterior facet tensor A^S and interior facet tensor $A^{S,0}$ on a mesh are obtained by discretizing the local contribution to a multilinear form on a cell, exterior facet or interior facet respectively.

The discrete operator A may be computed efficiently using an algorithm known as *assembly*, which is the topic of the next chapter. As we shall see then, an important tool is the *cell tensor* obtained as the discretization of the bilinear form on a local cell of the mesh. In particular, consider the discretization of a multilinear form that may be expressed as a sum of local contributions from each cell K of a mesh $\mathcal{T} = \{K\}$,

$$a(v_1, v_2, \dots, v_\rho; w_1, w_2, \dots, w_n) = \sum_{K \in \mathcal{T}} a_K(v_1, v_2, \dots, v_\rho; w_1, w_2, \dots, w_n).$$

This corresponds to the case $n_c = 1$, $n_f = n_f^0 = 0$ and $\Omega_1 = \Omega$ in (6.2). Discretizing a_K using the local finite element basis $\{\phi_j^{K,j}\}_{j=1}^{n_j}$ on K, we obtain the cell tensor

$$A_i^K = a_K(\phi_{i_1}^{K,1}, \phi_{i_2}^{K,2}, \dots, \phi_{i_\rho}^{K,\rho}; w_1, w_2, \dots, w_n). \tag{6.3}$$

The cell tensor A^K is a typically dense tensor of rank ρ and dimension $n_1 \times n_2 \times \dots \times n_\rho$. The discrete operator A may be obtained by appropriately summing the contributions from each cell tensor A^K. We return to this in detail below in Chapter [logg-3].

If $\Omega_k \subset \Omega$, the discrete operator A may be obtained by summing the contributions only from the cells covered by Ω_k. One may similarly define the exterior and interior facet tensors A^S and $A^{S,0}$ as the contributions from a facet on the boundary or the interior of the mesh. The exterior facet tensor A^S is defined as in (6.3) by replacing the domain of integration K by a facet S. The dimension of A^S is generally the same as that of A^K. The interior facet tensor $A^{S,0}$ is defined slightly differently by considering the basis on a *macro element* consisting of the two elements sharing the common facet S as depicted in Figure 6.2. For details, we refer to (Ølgaard et al., 2008).

6.3 The Action of a Multilinear Form

Consider the bilinear form

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

obtained from the discretization of the left-hand side of Poisson's equation. Here, v and u are a pair of test and trial functions. Alternatively, we may consider v to be a test function and u to be a given solution to obtain a *linear* form parametrized over the coefficient u,

$$(\mathcal{A}a)(v) \equiv a(v; u) = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x.$$

We refer to the linear form $\mathcal{A}a$ as the *action* of the bilinear form a. In general, the action of a ρ-linear form with n coefficients is a $(\rho - 1)$-linear form with $n + 1$ coefficients. In particular, the action of a bilinear form is a linear form, and the action of a linear form is a functional.

The action of a bilinear form plays an important role in the definition of matrix-free methods for solving differential equations. Consider the solution of a variational problem of the canonical form (6.1) by an Krylov subspace method such as GMRES (Generalized Minimal RESidual method) (Saad and Schultz, 1986) or CG (Conjugate Gradient method) (Hestenes and Stiefel, 1952). Krylov methods approximate the solution $U \in \mathbb{R}^N$ of the linear system $AU = b$ by finding an approximation for U in the subspace of $\mathbb{R}^N$ spanned by the vectors $b, Ab, A^2b, \ldots, A^kb$ for some $k \ll N$. These vectors may be computed by repeated application of the discrete operator A defined as above by

$$A_{ij} = a(\phi_i^1, \phi_j^2).$$

For any given vector $U \in \mathbb{R}^N$, it follows that

$$(AU)_i = \sum_{j=1}^N A_{ij} U_j = \sum_{j=1}^N a(\phi_i^1, \phi_j^2) U_j = a\left(\phi_i^1, \sum_{j=1}^N U_j \phi_j^2\right) = a(\phi_i^1; u_h),$$

where $u_h = \sum_{j=1}^N U_j \phi_j^2$ is the finite element approximation corresponding to the coefficient vector U. In other words, the application of the matrix A on a given vector U is given by the action of the bilinear form evaluated at the corresponding finite element approximation,

$$(AU)_i = (\mathcal{A}a)(\phi_i^2; u_h).$$

The variational problem (6.1) may thus be solved by repeated evaluation (assembly) of a linear form (the action $\mathcal{A}a$ of the bilinear form a) as an alternative to first computing (assembling) the matrix A and then repeatedly computing matrix–vector products with A. Which approach is more efficient depends on how efficiently the action may be computed compared to matrix assembly, as well as on available preconditioners. For a further discussion on the action of multilinear forms, we refer to (Bagheri and Scott, 2004).

6.4 The Derivative of a Multilinear Form

When discretizing nonlinear variational problems, it may be of interest to compute the derivative of a multilinear form with respect to one or more of its coefficients. Consider the nonlinear variational problem to find $u \in V$ such that

$$F(v; u) = 0 \quad \forall v \in V. \tag{6.4}$$

To solve this problem by Newton's method, we linearize $u = \bar{u} + \delta u$ around a fixed value $\bar{u}$ to obtain

$$0 = F(v; u) \approx F(v; \bar{u}) + F'(v; \bar{u})\delta u.$$

Given an approximate solution $\bar{u}$ of the nonlinear variational problem (6.4), we may then hope to improve the approximation by solving the linear variational problem

$$F'(v, \delta u; \bar{u}) \equiv F'(v, \bar{u})\delta u = -F(v; \bar{u}).$$

Here, F' is a bilinear form with two arguments v and δu, and one coefficient $\bar{u}$, and $-F$ is a linear form with one argument v and one coefficient $\bar{u}$.

When there is more than one coefficient, we will use the notation $\mathcal{D}_w$ to denote the derivative with respect to a specific coefficient w. In general, the derivative $\mathcal{D}$ of a ρ-linear form with $n > 0$ coefficients is a $(\rho+1)$-linear form with n coefficients. To solve the variational problem (6.4) using a matrix-free Newton method, we would thus need to repeatedly evaluate the linear form $(\mathcal{A}\mathcal{D}_u F)(v; \bar{u}_h, \delta u_h)$ for a given finite element approximation $\bar{u}_h$ and increment δu_h.

Note that one may equivalently consider the application of Newton's method to the nonlinear discrete system of equations obtained by a direct application of the finite element method to the variational problem (6.4) as discussed in Chapter [kirby7].

6.5 The Adjoint of a Bilinear Form

The adjoint a^* of a bilinear form a is the form obtained by interchanging the two arguments,

$$a^*(w, v) = a(v, w) \quad \forall v \in V^1 \quad \forall w \in V^2.$$

The adjoint of a bilinear form plays an important role in the error analysis of finite element methods as we saw in Chapter [kirby-7] and as will be discussed further in Chapter [massing] where we consider the linearized adjoint problem (the dual problem) of the general nonlinear variational problem (6.4). The dual problem takes the form

$$(\mathcal{D}_u F)^*(v, z; u) = \mathcal{D}_u M(v; u) \quad \forall v \in V,$$

or simply

$$F'^*(v, z) = M'(v) \quad \forall v \in V,$$

where $(\mathcal{D}_u F)^*$ is a bilinear form and $\mathcal{D}_u \mathcal{M}$ is a linear form (the derivative of the functional $\mathcal{M}$).

6.6 A Note on the Order of Test and Trial Functions

It is common in the literature to consider bilinear forms where the trial function u is the first argument, and the test function v is the second argument,

$$a = a(u, v).$$

With this notation, one is lead to define the discrete operator A as

$$A_{ij} = a(\phi_j^2, \phi_i^1).$$

In this book and throughout the code and documentation of the FEniCS Project, we have instead adopted the notation

$$a = a(v, u),$$

which leads to the more natural definition

$$A_{ij} = a(\phi_i^1, \phi_j^2),$$

which is particularly convenient when we consider multilinear forms of general arity. This ordering follows naturally from the ordering of dimensions for a matrix (rows before columns) and agrees with the fact that the rows of a matrix correspond to equations (test functions) and the columns to unknowns (expansion coefficients for the trial functions).

CHAPTER 7

Finite Element Assembly

By Anders Logg

Chapter ref: **[logg-3]**

Overview of general assembly algorithm.

CHAPTER 8

Quadrature Representation of Finite Element Variational Forms

By Kristian B. Ølgaard and Garth N. Wells

Chapter ref: **[oelgaard-2]**

Summarise work on optimizing quadrature representation using automated code generation and address automated selection of best representation.

CHAPTER 9

Tensor Representation of Finite Element Variational Forms

By Anders Logg and possibly others

Chapter ref: **[logg-4]**

Overview of tensor representation.

CHAPTER 10

Discrete Optimization of Finite Element Matrix Evaluation

By Robert C. Kirby, Matthew G. Knepley, Anders Logg, L. Ridgway Scott and Andy R. Terrel

Chapter ref: **[[kirby-4]]**

The tensor contraction structure for the computation of the element tensor A^K obtained in Chapter [logg-4] enables not only the construction of a compiler for variational forms, but an *optimizing* compiler. For typical variational forms, the reference tensor A^0 has significant structure that allows the element tensor A^K to be computed on an arbitrary element K in a reduced amount of arithmetic. Reducing the number of operations based on this structure leads naturally to several problems in discrete mathematics. This chapter introduces the kind of optimizations that are possible and discusses compile-time combinatorial optimization problems that form the core of the FErari project. (Kirby, Kirby and Logg, 2008, Kirby and Scott, 2007, Kirby et al., 2006)

We consider two basic kinds of optimizations in this chapter. First, we consider relations between pairs of rows in the reference tensor. This naturally leads to a graph that models proximity among these pairs in the sense that if two rows are ”close” together, then one may reuse results computed with the first row to compute with the second. This gives rise to a weighted graph that is (almost) a metric space, so we designate such optimizations ”topological”. Second, we consider relations between more than two rows of the reference tensor. Such relations typically rely on sets of rows, considered as vectors in Euclidean space, lying in a lower-dimensional space. Because we are using planes and hyperplanes to reduce the amount of computation, we describe these optimizations as ”geometric”.

10.1 Optimization Framework

The tensor paradigm developed in Chapter [logg-4] arrives at the representation

$$A_i^K = \sum_{\alpha \in \mathcal{A}} A_{i\alpha}^0 G_K^\alpha, \quad i \in \mathcal{I},$$

or simply

$$A^K = A^0 : G_K, \tag{10.1}$$

where $\mathcal{I}$ is the set of admissible multiindices for the element tensor A^K and $\mathcal{A}$ is the set of admissible multiindices for the geometry tensor G_K. The reference tensor A^0 can be computed at compile-time, and may then be contracted with a G_K to obtain the element tensor A^K for each cell K in the finite element mesh at run-time. The case of computing finite element stiffness matrices of size $n_K \times n_K$, where n_K is the dimension of the local finite element space on K, corresponds to $\mathcal{I}$ consisting of $|\mathcal{I}| = n_K^2$ multiindices of length two.

It is convenient to recast (10.1) in terms of a matrix–vector product:

$$A^0 : G_K \leftrightarrow \tilde{A}^0 \tilde{g}_K. \tag{10.2}$$

The matrix $\tilde{A}$ lies in $\mathbb{R}^{|\mathcal{I}| \times |\mathcal{A}|}$, and the vector $\tilde{g}_K$ lies in $\mathbb{R}^{|\mathcal{A}|}$. The resulting matrix–vector product can then be reshaped into the element tensor A^K. As this computation must occur for each cell K in a finite element mesh, it makes sense to try to make this operation efficient.

In the following, we will drop the subscripts and superscripts of (10.2) and consider the problem of computing

$$y = Ax$$

efficiently, where $A = \tilde{A}^0$ is a fixed matrix known *a priori* and $x = g_K$ is an arbitrary vector. We will study structure of A that enables a reduction in the amount of arithmetic required to form these products.

Before proceeding with the mathematical formulation, we give an example of a matrix A that we would like to optimize. In (10.3), we display the $6 \times 6 \times 2 \times 2$ reference tensor A^0 for computing a standard stiffness matrix discretizing the Laplacian with quadratic Lagrange elements on triangles. The rank four tensor is here depicted as a 6×6 matrix of 2×2 matrices. Full analysis would use a corresponding flattened 36×4 matrix A.

$$A^0 = \left(\begin{array}{cc|cc|cc|cc|cc|cc}
3 & 0 & 0 & -1 & 1 & 1 & -4 & -4 & 0 & 4 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 3 & 1 & 1 & 0 & 0 & 4 & 0 & -4 & -4 \\
\hline
1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\
1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\
\hline
-4 & 0 & 0 & 0 & -4 & -4 & 8 & 4 & 0 & -4 & 0 & 4 \\
-4 & 0 & 0 & 0 & 0 & 0 & 4 & 8 & -4 & -8 & 4 & 0 \\
\hline
0 & 0 & 0 & 4 & 0 & 0 & 0 & -4 & 8 & 4 & -8 & -4 \\
4 & 0 & 0 & 0 & 0 & 0 & -4 & -8 & 4 & 8 & -4 & 0 \\
\hline
0 & 0 & 0 & -4 & 0 & 0 & 0 & 4 & -8 & -4 & 8 & 4 \\
0 & 0 & 0 & -4 & -4 & -4 & 4 & 0 & -4 & 0 & 4 & 8
\end{array}\right) \tag{10.3}$$

10.2 Topological Optimization

It is possible to apply the matrix A depicted in (10.3) to an arbitrary vector x in fewer operations than a standard matrix–vector multiplication which requires 144 multiply–add pairs. This requires offline analysis of A and special-purpose code generation that applies the particular A to a generic x. For $A \in \mathbb{R}^{M\times N}$, let $\{a^i\}_{i=1}^M \subset \mathbb{R}^N$ denote the rows of A. The vector $y = Ax$ may then be computed by M dot products of the form $y_i = a^i x$. Below, we investigate relationships among the rows of A to find an optimized computation of the matrix–vector product.

For examination of A, we consider the following small subset of (10.3) which would only cost 40 multiply–add pairs but contains all the relations we use to optimize the larger version:

$$A = \begin{pmatrix}
a^1 \leftrightarrow A^0_{1,3} \\
a^2 \leftrightarrow A^0_{1,4} \\
a^3 \leftrightarrow A^0_{2,3} \\
a^4 \leftrightarrow A^0_{3,3} \\
a^5 \leftrightarrow A^0_{4,6} \\
a^6 \leftrightarrow A^0_{4,4} \\
a^7 \leftrightarrow A^0_{4,5} \\
a^8 \leftrightarrow A^0_{5,6} \\
a^9 \leftrightarrow A^0_{6,1} \\
a^{10} \leftrightarrow A^0_{6,6}
\end{pmatrix} = \begin{pmatrix}
1 & 1 & 0 & 0 \\
-4 & -4 & 0 & 0 \\
0 & 0 & 1 & 1 \\
3 & 3 & 3 & 3 \\
0 & 4 & 4 & 0 \\
8 & 4 & 4 & 8 \\
0 & -4 & -4 & -8 \\
-8 & -4 & -4 & 0 \\
0 & 0 & 0 & 0 \\
8 & 4 & 4 & 8
\end{pmatrix}. \tag{10.4}$$

A brief inspection of (10.4) shows that a^9 is zero; therefore, it does not need to be multiplied by the entries of x. In particular, if z entries of a^i are zero, then the dot product $a^i x$ requires $N - z$ multiply–add pairs rather than N.

Suppose $a^i = a^j$ for some $i \neq j$, as seen in the sixth and tenth rows of A. Then of course $y_i = y_j$, and only one dot product needs to be performed instead of two. In other cases, $\alpha a^i = a^j$ for some number α, as in the first and second rows of A. This means that after y_i has been computed, then $y_j = \alpha y_i$ may be computed with a single multiplication.

In addition to equality and colinearity as above, one may also consider other relations between the rows of A. A further inspection of A in (10.4) reveals rows that have some entries in common but are neither equal nor colinear. Such rows have a small *Hamming distance*, that is, the number of entries in which the two rows differ is small. This occurs frequently, as seen in, e.g., rows five and six . Write $a^j = a^i + (a^j - a^i)$. Then $a^j - a^i$ only has d nonzero entries, where d is the Hamming distance between a^i and a^j. Once y_i has been computed, one may thus compute y_j by

$$y_j = y_i + \left(a^j - a^i\right) x,$$

which requires only $d \leq N$ additional multiply–add pairs. If d is small compared to N, the savings are considerable.

In a recent article (Wolf and Heath, 2009), Heath and Wolf extend binary relations to include the partial colinearity of two vectors. For example, the sixth and seventh rows have parts that are colinear, namely $a^6_{2:4} = -a^7_{2:4}$. Such relationships reduce the computation of y_j to two multiply–add pairs; first a scaling of the result computed with y_j and then an additional multiplication with the non-matching entry in a^j.

All of these examples of structure either relate to a single row of A or else to a pair of rows of A. Such *binary* relations between pairs of rows are amenable to the formulation of graph-theoretic structures, as is developed in Section 10.3. Higher-order relations also occur between the rows of A. For example, the first and third rows may be added and scaled to make the fourth row. In this case, once a^1x and a^2x are known, the results may be used to compute a^4x using one addition and one multiplication, compared to four multiplications and three additions for direct evaluation of the dot product a^4x.

10.3 A Graph Problem

If we restrict consideration to binary relations between the rows of A, we are led naturally to a weighted, undirected graph whose vertices are the rows a^i of A. An edge between a^i and a^j with weight d indicates that if a^ix is known for some x, then that result may be used to compute a^jx with d multiply–add pairs. In practice, such edges also need to be labeled with information indicating the kind of relationship such as equality, colinearity or low Hamming distance.

To find the optimal computation through the graph, we use Prim's algorithm (**?**) for computing a minimum-spanning tree. A minimum spanning tree is a tree that connects all the vertices of the graph and has minimum total edge weight.

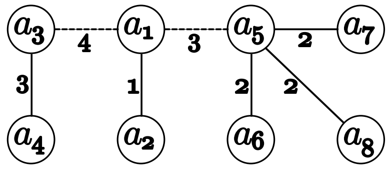


Figure 10.1: Minimum spanning tree (forest) for the vectors in (10.4). The dashed edges represent edges that do not reduce the number of operations and thus disconnect the graph.

In (**?**), it is demonstrated that, under a given set of relationships between rows, a minimum spanning tree in fact encodes an algorithm that optimally reduces arithmetic. This discussion assumes that the initial graph is connected. In principle, every a^i is no more than a distance of N away from any a^j. In practice, however, only edges $N - z$ are included in a graph since N is the cost of computing y_i without reference to y_j. This often makes the graph unconnected and thus one must construct a minimum spanning *forest* instead of a tree (a set of disjoint trees that together touch all the vertices of the graph). An example of a minimum spanning tree using the binary relations is shown in Figure 10.1.

Such a forest may then be used to determine an efficient algorithm for evaluating Ax as follows. Start with some a^i and compute $y^i = a^i x$ directly in at most N multiply–add pairs. The number of multiply–add pairs may be less than one if one or more entries of a^i are zero. Then, if a^j is a nearest neighbor of a^i in the forest, use the relationship between a^j and a^i to compute $y^j = a^j x$. After this, take a nearest neighbor of a^j, and continue until all the entries of y have been computed.

Additional improvements may be obtained by recognizing that the input tensor $G_K \leftrightarrow x$ is symmetric for certain operators like the Laplacian. In two space dimensions, G_K for the Laplacian is 2×2 with only 3 unique entries, and in three space dimensions it is 3×3 with only 6 unique entries. This fact may be used to construct a modified reference tensor A^0 with fewer columns. For other operators, it might have symmetry along some but not all of the axes.

Heath and Wolf proposed a slight variation on this algorithm. Rather than picking an arbitrary starting row a^i, they enrich the graph with an extra vertex labeled *IP* for "inner product." Each a^i is a distance $N - z$ from IP, where z is the number of vanishing entries in a^i. The IP vertex is always selected as the root of the minimum spanning tree. It allows for a more robust treatment of unary relations such as sparsity, and detection of partial colinearity relations.

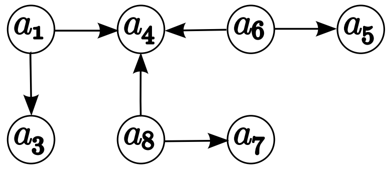


Figure 10.2: Generating graph for the vectors in (10.4).

10.4 Geometric Optimization

When relations between more than two rows are considered, the optimization problem no longer may be phrased in terms of a graph, but requires some other structure. In these cases, proving that one has found an optimal solution is typically difficult, and it is suspected that the associated combinatorial problems are NP-hard.

As a first attempt, one can work purely from linear dependencies among the data as follows. Let $B = \{b^i\}_i \subseteq \{a^i\}_{i=1}^N$ be a maximal set of nonzero rows of A such that no two rows are colinear. Then enumerate all triples which are linearly dependent,

$$T = \left\{ \left\{ b^i, b^j, b^k \right\} \subseteq B : \exists\, \alpha_1, \alpha_2, \alpha_3 \neq 0 : \alpha_1 b^i + \alpha_2 b^j + \alpha_3 b^k = 0 \right\}.$$

The idea is now to identify some subset C of B that may be used to recursively construct the rest of the rows in B using the relationships in T.

Given some $C \subset B$, we may define the *closure* of C, denoted by $\bar{C}$, as follows. First of all, if $b \in C$, then $b \in \bar{C}$. Second, if $b \in B$ and there exist $c, d \in \bar{C}$ such that $\{b, c, d\} \in T$, then $b \in \bar{C}$ as well. If $\bar{C} = B$, we say that C is a *generator* for B or that C *generates* B.

The recursive definition suggests a process for constructing the closure of any set C. In the course of constructing the closure, one may also construct a directed, acyclic graph that indicates the linear dependence being used. Each $b \in C$ will have no out-neighbors, while each $b \in \bar{C} \backslash C$ will point to two other members of $\bar{C}$. This graph is called a *generating graph*. Using (10.4), we have the following sets B, T, and C, with the generating graph shown in Figure 10.2:

$$\begin{array}{rcl} B & = & \{a_1, a_3, a_4, a_5, a_6, a_7, a_8\} \\ T & = & \{(a_1, a_3, a_4), (a_4, a_5, a_6), (a_6, a_7, a_8)\} \\ C & = & \{a_3, a_4, a_5, a_7\} \end{array}$$

If C generates B, then the generating graph indicates an optimized (but perhaps not optimal) process for computing $\{y^i = b^i x\}_i$. Take a topological ordering of the vectors b^i according to this graph. Then, for each b^i in the topological ordering, if b^i has no out neighbors, then $b^i x$ is computed explicitly. Otherwise, b^i will point to two other vectors b^j and b^k for which the dot products with x will already

triangles					tetrahedra				
degree	M	N	MN	MAPs	degree	M	N	MN	MAPs
1	6	3	18	9	1	10	6	60	27
2	21	3	63	17	2	55	6	330	101
3	55	3	165	46	3	210	6	1260	370

Table 10.1: Number of multiply–add pairs for graph-optimized Laplace operator (MAPS) compared to the basic number of multiply–add pairs (MN).

be known. Since the generating graph has been built from the set of linearly dependent triples T, there must exist some β_1, β_2 such that $b^i = \beta_1 b^j + \beta_2 b^k$. We may thus compute y^i by

$$y^i = b^i x = \beta_1 b^j x + \beta_2 b^k x,$$

which requires only two multiply–add pairs instead of N.

To make best use of the linear dependence information, one would like to find a generator C that has as few members as possible. We say that a generator C is *minimal* for B if no $C' \subset C$ also generates B. A stronger requirement is for a generator to be *minimum*. A generator C is minimum if no other generator C' has lower cardinality. Heuristics for constructing minimal generators are considered in (**?**), and it is not currently known whether such heuristics construct minimum generators or how hard the problem of finding minimum generators is.

Given a minimal generator C for B, one may consider searching for higher order linear relations among the elements of C, such as sets of four items that have a three-dimensional span. The discussion of generating graphs and their utilization is the same in this case.

Wolf and Heath (Wolf and Heath, 2009) study combining binary and higher-order relations between the rows of A in a hypergraph model. While greedy algorithms provide optimal solutions for a graph model, they demonstrate that the obvious generalizations to hypergraphs can be suboptimal. While the hypergraph problems are most likely very hard, they develop heuristics that perform well and provide additional optimizations beyond the graph models. So, even if a non-optimal solution is found, it still provides improved reduction in arithmetic requirements.

In Table 10.4, topological and geometric optimization is compared for the Laplacian using quadratic through quartic polynomials on tetrahedra. In the geometric case, the vectors a^i were filtered for unique direction, i.e., only one vector for each class of colinear vectors was retained. Then, a generating graph was constructed for the remaining vectors using pairwise linear dependence. The generator for this set was then searched for linear dependence among sets of four vectors, and a generating graph constructed. Perhaps surprisingly, the geometric optimization found flop reductions competitive with or better than graph-based binary relations. These are shown in Table 10.4.

	2	3	4
topological	101	370	1118
geometric	105	327	1072

Table 10.2: Comparison of topological and geometric optimizations for the Laplace operator on tetrahedra using polynomial degrees two through four. In each case, the final number of MAPs for the optimized algorithm is reported. The case $q = 1$ is not reported since then both strategies yield the same number of operations.

10.5 Optimization by Dense Linear Algebra

As an alternative to optimizations that try to find a reduced arithmetic for computing the element tensor A^K, one may consider computing the element tensor by efficient dense linear algebra. As above, we note that the entries of the element tensor A^K may be computed by the matrix–vector product $\tilde{A}^0 \tilde{g}_K$. Although zeros may appear in $\tilde{A}^0$, this is typically a dense matrix and so the matrix–vector product may be computed efficiently with Level 2 BLAS, in particular using a call to `dgemv`. There exist a number of optimized implementations of BLAS, including hand-optimized vendor implementations, empirically and automatically tuned libraries (**?**) and formal methods for automatic derivation of algorithms (**?**).

The computation of the element tensor A^K may be optimized further by recognizing that one may compute the element tensor for a batch of elements $\{K_i\}_i \subset \mathcal{T}$ in one matrix–matrix multiplication:

$$\begin{bmatrix} \tilde{A}^0 \tilde{g}_{K_1} & \tilde{A}^0 \tilde{g}_{K_2} & \cdots \end{bmatrix} = \tilde{A}^0 \begin{bmatrix} \tilde{g}_{K_1} & \tilde{g}_{K_2} & \cdots \end{bmatrix}.$$

This matrix-matrix product may be computed efficiently using a single Level 3 BLAS call (`dgemm`) instead of a sequence of Level 2 BLAS calls, which typically leads to better floating-point performance.

10.6 Notes on Implementation

A subset of the optimizations discussed in this chapter are available as part of the FErari Python module. The current version of FErari (0.2.0) implements optimization based on finding binary relations between the entries of the element tensor. With optimizations turned on, FFC calls FErari at compile-time to generate optimized code. Optimization for FFC can be turned on either by the `-O` parameter when FFC is called from Python, or by setting

```
  parameters["form_compiler"]["optimization"] = True
```

when FFC is called as a just-in-time compiler from the DOLFIN Python interface. Note that the FErari optimizations are only used when FFC generates code based on the tensor representation described in Chapter [logg-4]. When FFC generates code based on quadrature, optimization is handled differently, as described in Chapter [oelgaard-2].

CHAPTER 11

Parallel Adaptive Mesh Refinement

By Niclas Jansson, Johan Hoffman and Johan Jansson

Chapter ref: **[hoffman-4]**

For many interesting problems one is often interested in rather localized features of a solution, for example separation or transition to turbulence in flow problems. It is often the case that it is to expensive to uniformly refine a mesh to such an extent that these features develops. The solution is to only refine the mesh in the regions of most interest, or for example where the error is large.

This chapter is based on the work in (Jansson, 2008). First a brief summary of the theory behind mesh refinement is given, followed by a discussion of the issues with parallel refinement together with our proposed solution. The chapter ends with a presentation of our implementation, of a fully distributed parallel adaptive mesh refinement framework on a Blue Gene/L.

11.1 A brief overview of parallel computing

In this chapter, parallel computing refers to distributed memory systems with message passing. It is assumed that the system is formed by linking compute nodes together with some interconnect, in order to form a virtual computer (or cluster) with a large amount of memory and processors.

It is also assumed that the data, the mesh in this case is distributed across the available processors. Shared entities are duplicated or "ghosted" on adjacent processors. In order to fully represent the original mesh from the smaller distributed sub meshes, it is essential that the information for the shared entities are the same on all processors.

11.2 Local mesh refinement

Local mesh refinement has been studied by several authors over the past years. The general idea is to bisect an element into a set of new ones, with the constraint that the produced mesh must be valid. A mesh is usually valid if there are no “hanging nodes”, that is no node should be on another element’s facet. How element should be bisected in order to ensure this differs between different methods. One common theme is edge bisection.

A common edge bisection algorithm bisects all edges in an element, introducing a new vertex on each edge, and connecting them together to form the new elements (see for example (Bey, 1995)). Other methods bisects only one of the edges, which edge depends on the method. For example one could select the edge which where most recently refined, this method is often referred to as the newest vertex approach and where described in (Bänsch, 1991). Another popular edge bisection method is the longest edge (Rivara, 1984), where one always select the longest edge for refinement. In order to ensure that the mesh is free of “hanging nodes”, the algorithm recursively bisects elements until there are no “hanging nodes” left.

11.2.1 The challenge of parallel mesh refinement

Performing the refinement in parallel adds additional constraints on the refinement method. Not only should the method prevent “hanging nodes”, it must also be guaranteed to terminate in a finite number of steps.

In the parallel case, each processor owns a small part of the distributed mesh. So if a new vertex is introduced on the boundary between two processors, the algorithm must ensure that the information propagates to all neighboring processors.

For an algorithm that bisects all edges in an element, the problem reduces to a global decision problem, deciding which of the processors information that should be used on all the other processors. But for an algorithm that propagates the refinement like the longest edge, the problem becomes a set of synchronization problems i) to detect and handle refinement propagation between different processors and ii) to detect global termination.

The first problem could easily be solved by dividing the refinement into two different phases, one local refinement phase and one propagation phase. In the first phase elements on the processor are refined with an ordinary serial refinement method. This could create non conforming elements on the boundary between processors. These are fixed by propagating the refinement to the neighboring processor. This is done in the second propagation phase. But since the next local refinement phase could create an invalid mesh, one could get another propagation phase and the possibility of another and so forth. However, if the longest edge algorithm is used, termination is guaranteed (Castaños and Savage, 1999).

But the problem is to detect when all these local meshes are conforming, and also when they are conforming at the same time. That means one has to detect global termination, which is a rather difficult problem to solve efficiently, especially for massively parallel systems for which we are aiming.

There has been some other work related to parallel refinement with edge bisection. For example a parallel newest vertex algorithm has been done by Zhang (Zhang, 2005). Since the algorithm does not need to solve the termination detection problem, scaling is simply a question of interconnect latency. Another work is the parallel longest edge algorithm done by Castaños and Savage (Castaños and Savage, 1999). They solve the termination detection problem with Dijkstras general distributed termination algorithm, which simply detects termination by counting messages sent and received from some controlling processor. However, in both of these methods they only used a fairly small amount of processors, less then one hundred, so it is difficult to estimate how efficient and scalable these algorithms are. For more processors, communication cost and concurrency of communication patterns starts to become important factors. Therefore, we tried to design an algorithm that would scale well for thousands of processors.

11.2.2 A modified longest edge bisection algorithm

Instead of trying to solve the termination detection problem, one could try to modify the refinement algorithm in such a way that it would only require one synchronization step, thus less communication. With less communication overhead it should also scale better for a large number of processors.

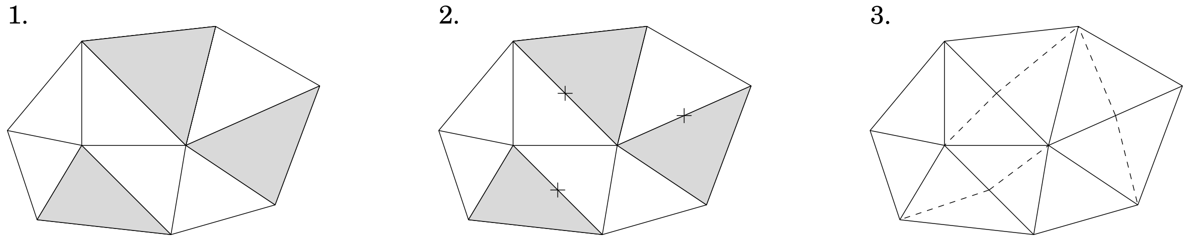


Figure 11.1: An example of the refinement algorithm used. First a set of elements are marked for refinement (1). The longest edges are found (2), and all elements connected to these edges are finally bisected, the dashed lines in (3).

When this work started, DOLFIN did not have a pure longest edge implementation. Instead of recursively fixing “hanging nodes” it bisected elements in pairs (or groups) (see figure 11.1). Since this algorithm always terminate the refinement by bisecting all elements connected to the refined edge, it is a perfect candidate for an efficient parallel algorithm. Since, if the longest edge is shared by different processors, the algorithm must only propagate the refinement onto all elements (processors) connected to that edge, but then no further propagation is possible (see figure 11.2).

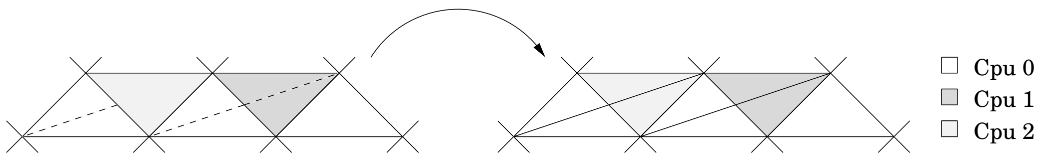


Figure 11.2: An example of the two simple cases of propagation. The dashed lines refers to how processors wants to bisect the elements.

However, notifying an adjacent processor of propagation does not solve the problem entirely. As mentioned in section 11.1, all mesh entities shared by several processors must have the same information in order to correctly represent the distributed mesh. The refinement process must therefore guarantee that all newly created vertices are assigned the same unique information on all the neighboring processors. Another problematic case arises when processors refine the same edge and the propagation “collides” (see figure 11.2). In this case the propagation is done implicitly but the processors must decide which new information to use.

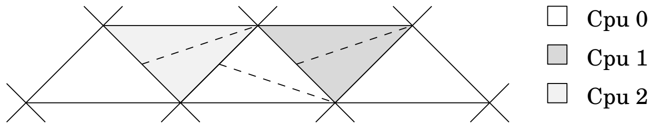


Figure 11.3: An example of the problematic case with multiple propagations. The dashed lines refers to how processors wants to bisect the elements.

A more complicated case is when an element receives multiple propagation (possibly from different processors) on different edges (see figure 11.3). Since the modified longest edge algorithm only allows one edge to be bisected per element, one of the propagations must be selected and the other one rejected. This however adds a difficulty to the simple algorithm. How should the processors decide upon which edge to be refined? Clearly this could not be done arbitrarily, since when a propagation is forbidden, all refinement done around that edge must be removed. Thus, in the worst case it could destroy the entire refinement.

To solve the edge selection problem perfectly one needs an algorithm with a global view of the mesh. In two dimensions with a triangular mesh, the problem could be solved rather simple since each propagation could only come from two different edges (one edge is always facing the interior). By exchanging the desired propagation edges processors could match theirs selected edges with the propagated ones, in an attempt to minimize the number of forbidden propagations. However, in three dimensions the problem starts to be such complicated that multiple exchange steps are needed in order to solve the problem. Hence, it becomes too expensive to solve.

Instead we propose an algorithm which solves the problem using an edge voting scheme. Each processor refines the boundary elements, find their longest edge and cast a vote for it. These votes are then exchanged between processors, which add the received votes to its own set of votes. Now the refinement process restarts, but instead of using the longest edge criteria, edges are selected depending on the maximum numbers of votes. In the case of a tie, the edge is selected depending on a random number assigned to all votes.

Once a set of edges has been selected from the voting phase, the actually propagation starts by exchanging these with the other processors. However, the voting could fail to "pair" refinements together. For example, an element could lie between two processors which otherwise does not share any common face. Each of these processors wants to propagate into the neighboring element but on different edges (see figure 11.4). Since the processors on the left and right side of the element do not receive edge votes from each other, the exchange of votes will in this case not help with selecting an edge that would work for both processors.

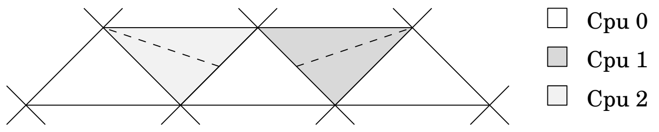


Figure 11.4: An example of the case when edge votes could be missed. The dashed lines refers to how processors wants to bisect the elements.

To fix this, an additionally exchange step is needed and maybe another and so forth, rendering the perfect fix impossible. Instead, the propagation step ends by exchanging the refined edges which gave rise to a forbidden propagation. All processors could then remove all refinements that these edges introduced, and in the process, remove any hanging nodes on the boundary between processors.

11.3 The need of dynamic load balancing

For parallel refinement, there is an additional problem not present in the serial setting. As one refines the mesh, new vertices are added arbitrarily at any processor. Thus, rendering an initially good load balance useless. Therefore, in order to sustain a good parallel efficiency the mesh must be repartitioned and redistributed after each refinement, in other words dynamic load balancing is needed.

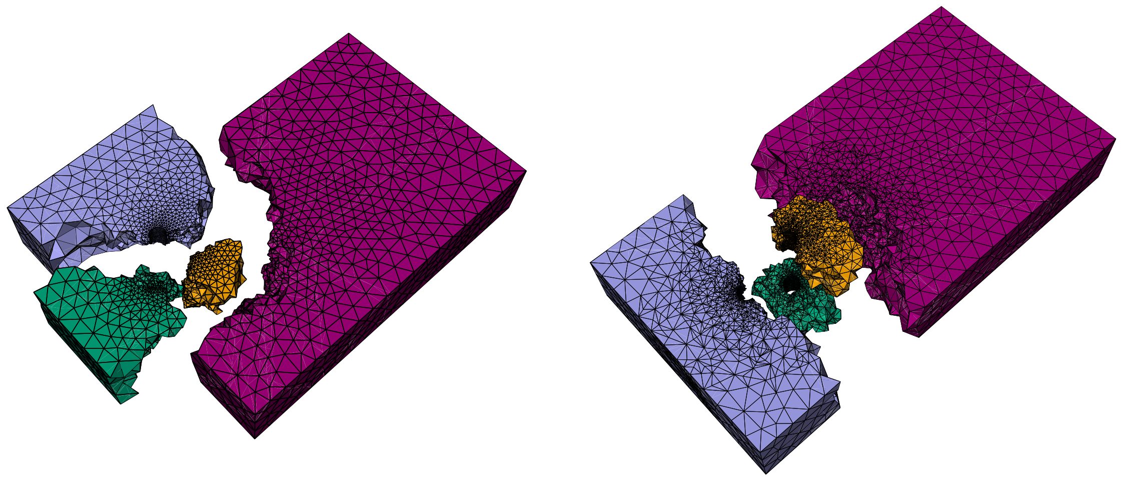


Figure 11.5: An example of load balancing were the locality of the data is considered. The mesh is refined three times around the cylinder, and for each step load balancing is performed, shading refers to processor id.

In the worst case, the load balancing routine must be invoked every time a mesh is adapted, so it has to be rather efficient, and for our aim, scale well for a large number of processors. There are mainly two different load balancing methods used today, diffusive and remapping methods. Diffusive methods, like the physical meaning, by finding a diffusion solution a heavily loaded processor's vertices would move to another processor, and in that way smear out the imbalance, described for example in (Hu and Blake, 1995, Schloegel et al., 1997). Remapping methods relies on the partitioner's efficiency of producing good partitions from an already partitioned dataset. In order to avoid costly data movement, a remapping method tries to assign the new partitions to processors in an optimal way. For problems where the imbalance occurs rather localized, the remapping methods seems to perform better (Schloegel et al., 1998). Hence, it maps perfectly to the localized imbalance caused by local mesh refinement.

In this work, we used the load balancing framework of PLUM (Oliker, 1998) a remapping load balancer. The mesh is repartitioned according to an imbalance model. Repartitioning is done before refinement, this would in theory minimize data movement and speedup refinement, since a more balanced number of element would be bisected on each processor.

11.3.1 Workload modelling

The workload is modelled by a weighted dual graph of the mesh. Let $G = (V, E)$ be the dual graph of the mesh, q be one of the partitions and let w_i bet the computational work (weights) assigned to each graph vertex. The processor workload is then defined as

$$W(q) = \sum_{\forall w_i \in q} w_i \tag{11.1}$$

where communication costs are neglected. Let W_{avg} be the average workload and W_{max} be the maximum, then the graph is considered imbalanced if

$$W_{\text{max}}/W_{\text{avg}} > \kappa \tag{11.2}$$

where the threshold value κ is based on the problem or machine characteristics.

This model suits the modified longest edge algorithm (section 11.2.2) perfectly. Since the modifications reduces the algorithm to only have one propagation and/or synchronization step. The workload calculation becomes a local problem, thus it is rather cheap to compute. So if we let each element represent one unit of work, a mesh adapted by the modified algorithm would produce a dual graph with vertex weights equal to one or two. Each element is only bisected once, giving a computational weight of two elements for each element marked for refinement.

11.3.2 Remapping strategies

Remapping of the new partitions could be done in numerous ways, depending on what metric one tries to minimize. Usually one often talks about the metrics TOTALV and MAXV. MAXV tries to minimize the redistribution time by lowering the flow of data, while TOTALV lowers the redistribution time by trying to keep the largest amount of data local, for a more detailed description see (Oliker, 1998). We have chosen to focus on the TOTALV metric, foremost since it much cheaper to solve then MAXV, and it also produces equally good (or even better) balanced partitions.

Independent of which metric one tries to solve. The result from the repartitioning is placed in a similarity matrix S, where each entry $S_{i,j}$ is the number of vertices on processor i which would be placed in the new partition j. In our case, we want to keep the largest amount of data local, hence to keep the maximum row entry in S local. This could be solved by transforming the matrix S into a bipartite graph where each edge $e_{i,j}$ is weighted with $S_{i,j}$, the problem then reduces to the maximally weighted bipartite graph problem (Oliker, 1998).

11.4 The implementation on a massively parallel system

The adaptive refinement method described in this chapter was implemented using an early parallel version of DOLFIN, for a more detailed description see

(Jansson, 2008). Parallelization was implemented for a message passing system, and all the algorithms were designed to scale well for a large number of processors.

The challenge of implementing a refinement algorithm on a massively parallel system is as always the communication latency. In order to avoid that the message passing dominates the runtime, it is important that the communication is kept at a minimum. Furthermore, in order to obtain a good parallel efficiency, communication patterns must be design in such way that they are highly concurrent, reducing processors idle time etc.

11.4.1 The refinement method

Since element bisection is a local problem, without any communication, the only part of the refinement algorithm that needs to be well designed for a parallel system is the communication pattern, used for refinement propagation.

For small scale parallelism, one could often afford to do the naive approach, loop over all processors and exchange messages without any problem. When the number of processors are increased, synchronization, concurrency and deadlock prevention starts to become important factors to considered when designing the communication pattern. A simple and reliable pattern is easily implemented as follows. If the processors send data to the left and receive data from the right in a circular pattern, all processors would always be busy sending and receiving data, thus no idle time.

Algorithm 1 Communication pattern

```
for i =1 to p-1 do
    src ← (rank - 1 + p) mod p
    dest ← (rank + 1) mod p
    sendrecv(send sendbuff(dest) to dest and recv from src)
end for
```

The refinement algorithm outlined in 11.2.2 is easily implemented as a loop over all elements marked for refinement. For each marked element it finds the longest edge and bisect all elements connected to that edge. However, since an element is only allowed to be bisected once, the algorithm is only allowed to select the longest edge which is part of an unrefined element. In order to make this work in parallel, one could structure the algorithm in such a way that it first refines the shared elements, and propagate the refinements. After that it could refine all interior elements without any communication.

If we let $\mathcal{B}$ be the set of elements on the boundary between processors, $\mathcal{R}$ the set of elements marked for refinement. Then by using algorithm 1 we could efficiently express the refinement of shared entities in algorithm 2 with algorithm 3.

Algorithm 2 Refinement algorithm

```
refine shared entities
for all $c \in \mathcal{R}$ do
    find longest edge $e$ not marked as forbidden
    for all elements $c_1$ connected to $e$ do
        bisect element $c_1$
    end for
end for
```

Algorithm 3 Refinement of shared entities

```
for all $c \in \mathcal{B} \cup \mathcal{R}$ do
    find longest edge $e$
    if $e$ is on the boundary then
        vote($e$) ← vote($e$) + 1
    end if
end for
exchange votes with algorithm 1
mark all elements $\in \mathcal{B}$ with the maximum number of votes for refinement
for all received votes on edge $e$ do
    increase vote($e$)
end for
for all $c \in \mathcal{B}$ do
    mark $\max_{e \in c}$(vote($e$)) for refinement
end for
exchange refinement with algorithm 1
for all received refinement do
    if $e$ is not refined and not part of a refined element then
        mark edge and propagate refinement
    else
        send back illegal propagation
    end if
end for
for all received illegal propagations do
    remove refinement and hanging nodes
end for
```

11.4.2 The remapping scheme

The maximally weighted bipartite graph problem for the TOTALV metric could be solved in an optimal way in $O(V^2 \log V + VE)$ steps (Oliker, 1998). Recall that the vertices in the graph are the processors. Calculating the remapping could therefore become rather expensive if a large number of processors are used. Since the solution does not need to be optimal, a heuristic algorithm with a runtime of $O(E)$ was used.

The algorithm starts by generating a sorted list of the similarity matrix S. It then steps through the list and selects the largest value which belongs to an unassigned partition. It was proven in (Oliker, 1998) that the algorithm's solution is always greater than half of the optimal solution, thus it should not be a problem to solve the remapping problem in a sub optimal way. Sorting was implemented (as in the original PLUM paper) by a serial binary radix sort (the matrix S were gathered onto one processor), performing β passes and using 2^r "buckets" for counting. In order to save some memory the sorting was performed per byte of the integer instead of the binary representation. Since each integer is represented by 4 bytes (true even for most 64-bits architectures) the number of passes required was $\beta = 4$, and since each byte have 8 bits the number of "buckets" needed were 2^8.

However, since the similarity matrix S is of the size $P \times P$ where P is the number of processors, the sorting will have a runtime of $O(P^2)$. This should not cause any problem on a small or medium size parallel computer, as the one used in the fairly old PLUM paper. But after 128-256 processors the complexity of the sorting starts to dominates in the load balancer. To solve this problem, instead of sorting S on one processor we implemented a parallel binary radix sort. The unsorted data of length N was divided into N/P parts which were assigned to the available processors. The internal β sorting phases were only modified with a parallel prefix calculation and a parallel swap phase (when the elements are moved to the new "sorted" locations).

Algorithm 4 Parallel radix sort

```
for i = 0 to $\beta$ do
    for j = 0 to N do
        count[i byte of data(j)] $\leftarrow$ count[i byte of data(j)] + 1
    end for
    count $\leftarrow$ Allreduce(count)
    for j = 0 to $2^r$ do
        index(j) $\leftarrow$ ParallelPrefix(count(j))
    end for
    redistribute elements according to index
end for
```

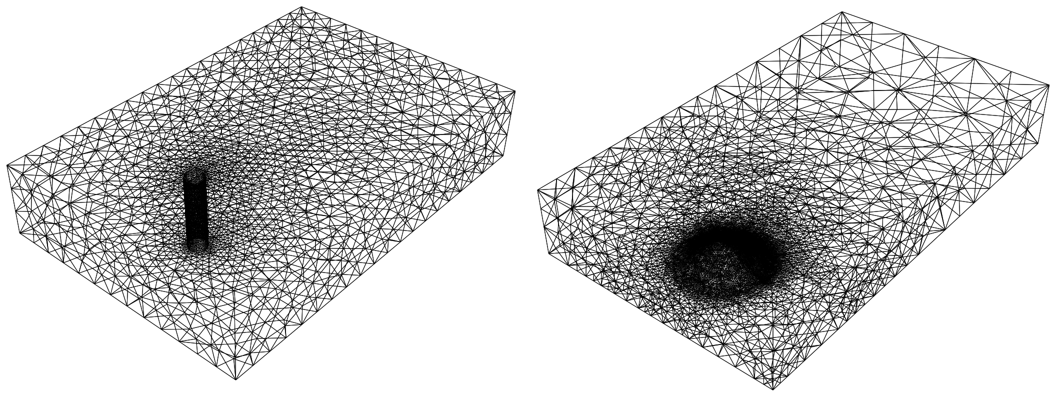


Figure 11.6: Two of the meshes used during evaluation.

11.4.3 Theoretical and experimental analysis

The adaptive refinement method described in this chapter has been successfully tested on a 1024 node Blue Gene/L, each dual-processor node could either be used to run two separate programs (virtual mode) or run in coprocessor mode where one of the processor works as an offload engine, for example handling parts of the message passing stack (Gara et al., 2005, Moreira et al., 2005).

As a test problem we used an unstructured mesh and refined all elements inside a small local region, timed mesh refinement and load balancing times. The problem were tested on $P = 32, 128, 512$ nodes both in virtual and coprocessor mode. Since the smallest number of nodes that could be allocated was 32, all speedup results are presented as a comparison against the time for 32 processors. To capture possible communication bottlenecks, three different unstructured meshes were used. First a cylinder mesh with $n_v = 566888$ vertices, secondly a hill with $n_v = 94720$ vertices and finally, the cylinder again but with $n_v = 1237628$ vertices instead.

The regions selected for refinement were around the cylinder and behind the hill. Since these regions already had the most number of elements in the mesh, refinement would certainly result in an workload imbalance. Hence, trigger the load balancing algorithms. In order to analyze the experimental results we used a performance model which decompose the total runtime T into one serial computational cost T_{comp}, and a parallel communication cost T_{comm}.

$$T = T_{\text{comp}} + T_{\text{comm}} \tag{11.3}$$

The mesh refinement has a local computational costs consisting of iterating over and bisecting all elements marked for refinement, for a mesh with N_c elements $O(N_c/P)$ steps. Communication only occurs when the boundary elements needs to be exchanged. Thus, each processor would in the worst case communicate with $(P-1)$ other processors. If we assume that there are N_s shared edges, the total runtime with communication becomes,

$$T_{\text{refine}} = O\left(\frac{N_c}{P}\right)\tau_f + (P-1)(\tau_s + N_s\tau_b) \tag{11.4}$$

where τ_f is the time to perform one (floating point) operation, τ_s is the latency and τ_b the bandwidth. So based on the performance model, more processors would lower the computational time, but in the same time increase the communication time.

The most computationally expensive part of the load balancer is the remapping or assignment of the new partitions. As discussed earlier, we used an heuristic with a runtime of $O(E)$, the number of edges in the bipartite graph. Hence, in the worst case $E \approx P^2$. The sorting phase is linear, and due to the parallel implementation it runs in $O(P)$. Communication time for the parallel prefix calculation is given by, for m data it sends and calculates in m/P steps. Since the prefix consists of 2^r elements, it would take $2^r/P$ steps, and we have to do this for each β sorting phases. In the worst case the reordering phase (during sorting) needs to send away all the elements, thus P communication steps, which gives the total runtime.

$$T_{\text{loadb}} = O(P^2)\tau_f + \beta\left(\tau_s + \left(\frac{2^r}{P} + P\right)\tau_b\right) \tag{11.5}$$

According to this, load balancing should not scale well for a large number of processors (due to the $O(P^2)$ term). However, the more realistic average case should be $O(P)$. So again, with more processors the runtime could go up due to communication costs.

If we then compare with the experimental results presented in figure 11.7, we see that the performance degenerates when a large number of processors are used. The question is why? Is it solely due to the increased communication cost? Or is the load balancer's computational cost to blame?

First of all, one could observe that when the number of elements per processor is small. The communication costs starts to dominate, see for example the results for the smaller hill mesh (represented by a triangle in the figure). The result is better for the medium size cylinder (represented by a diamond), and even better for the largest mesh (represented by a square). If the load balancing time was the most dominating part, a performance drop should have been observed around 128 - 256 processors. Instead performance generally drops after 256 processors. A possible explanation for this could be the small amount of local elements. Even for the larger 10^6 element mesh, with 1024 processors the number of local elements is roughly 10^3, which is probably too few to balance the communication costs.

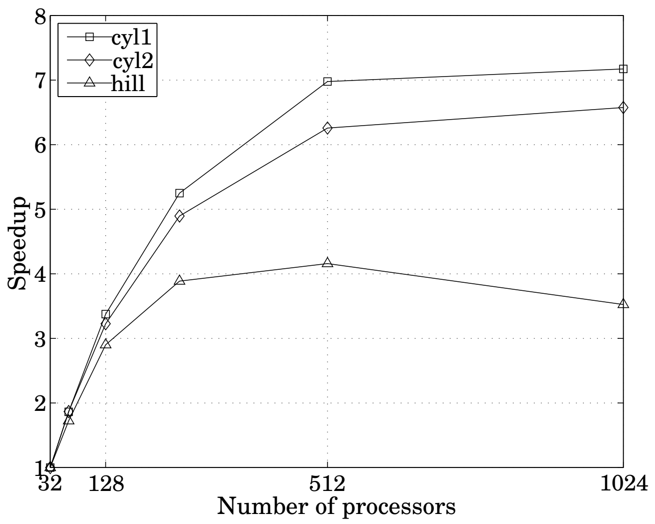


Figure 11.7: Refinement speedup (incl. load balancing)

Processors	Execution time	Speedup
256	33.50	1.0
512	23.19	1.44
1024	19.24	1.74

Table 11.1: Refinement time for a nine million vertices mesh

This illustrate the curse of massively parallel, distributed memory systems. In order to gain anything from the large amount of processors, either one has to have a large local computational cost, or one has to increase the problem size in order to mask the communication latency. To illustrate how large the mesh needs to be, we uniformly refined the larger cylinder, locally refined the same region as before and timed it for 256,512 and 1024 processors. Now the refinement performs better, as can be seen in table 11.1. But again performance drops between 512 and 1024 processors.

However, despite the decrease in speedup, one could see that the algorithm seems to scale well, given that the local amount of elements are fairly large. It is close to linear scaling from 32 to 256 processors, and we see no reason for it to not scale equally good between 256 and 1024 processors, given that the local part of the mesh is large enough.

11.5 Summary and outlook

In this chapter we presented some of the challenges with parallel mesh refinement. How these could be solved and finally how these solutions were implemented for a distributed memory system. In the final section some theoretical and experimental performance results were presented and explained. One could observe how well the implementation performs, until the curse of slow interconnect starts to affect the results.

One aspect of the refinement problem that has not been touched upon in this chapter is the mesh quality. The modification done to the longest edge algorithm (see section 11.2.2), unfortunately destroys the good properties of the original recursive algorithm. It was a trade off between efficiency and mesh quality. As mentioned before, the problem with the original algorithm is the efficiency of propagation and global termination detection. Currently our work is focused in overcoming these problems, implementing an efficient parallel refinement method with good quality aspects, which also performs well for thousands of processors.

Part II

Implementation

CHAPTER 12

DOLFIN: A C++/Python Finite Element Library

By Anders Logg and Garth N. Wells

Chapter ref: **[logg-2]**

Overview and tutorial of DOLFIN.

CHAPTER 13

FFC: A Finite Element Form Compiler

By Anders Logg and possibly others

Chapter ref: **[logg-1]**

► Editor note*: Oelgaard/Wells might possibly write something here on FFC quadrature evaluation, or it will be included in a separate chapter. Marie will possibly write something here on H(div)/H(curl), or it will be included in a separate chapter.*

Overview and tutorial of FFC.

CHAPTER 14

FErari: An Optimizing Compiler for Variational Forms

By Robert C. Kirby and Anders Logg

Chapter ref: **[kirby-3]**

We describe the implementation of an optimizing compiler for variational forms based on the discrete optimization methodology described in an earlier chapter. The major issues are communicating reference tensors from FFC to the optimizer, FErari, performing the actual optimization, and communicating abstract syntax for the optimized algorithm back to FFC for code generation. We include some results indicating the kinds of speedups that may be achieved and what factors influence the effectiveness of the optimizations.

CHAPTER 15

FIAT: Numerical Construction of Finite Element Basis Functions

By Robert C. Kirby

Chapter ref: **[kirby-2]**

15.1 Introduction

The FIAT project (**??**) implements the mathematical framework described in Chapter **??** as a Python package, working mainly in terms of numerical linear algebra. Although an implementation floating-point arithmetic presents some challenges relative to symbolic computation, it can allow greater efficiency and consume fewer resources, especially for high order elements. To obtain efficiency in Python, the compute-intensive operations are expressed in terms of numerical linear algebra and performed using the widely distributed `numpy` package.

The FIAT project is one of the first FEniCS projects, providing the basis function back-end for `ffc` and enabling high-order H^1, $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements. It is widely distributed, with downloads on every inhabited continent and in over sixty countries, averaging about 100 downloads per month.

This chapter works in the context of a Ciarlet triple (K, P, N) (**?**), where K is a fixed reference domain, typically a triangle or tetrahedron. P is a finite-dimensional polynomial space, though perhaps vector- or tensor-valued and not coincident with polynomials of some fixed degree. $N = \{\ell_i\}_{i=1}^{|P|}$ is a set of linear functionals spanning P'. Recalling Chapter (**?**), the goal is first to enumerate a convenient basis $\{\phi_i\}_{i=1}^{|P|}$ for P and then to form a generalized Vandermonde system

$$VA = I,$$

where $V_{ij} = \ell_i(\phi_j)$. The columns of $A = V^{-1}$ store the expansion coefficients of the nodal basis for (K, P, N) in terms of the basis $\{\phi_i\}$.

15.2 Prime basis: Collapsed-coordinate polynomials

High order polynomials in floating-point arithmetic require stable evaluation algorithms. FIAT uses the so-called collapsed-coordinate polynomials (**?**) on the triangle and tetrahedra. Let $P_i^{\alpha,\beta}(x)$ denote the Jacobi polynomial of degree i with weights α and β. On the triangle K with vertices $(-1, 1)$, $(1, -1)$, $(-1, 1)$, the polynomials are of the form

$$D^{p,q}(x, y) = P_p^{0,0}(\eta_1) \left(\frac{1-\eta_2}{2}\right)^i P_j^{2i+1,0}(\eta_2)$$

where

$$\begin{aligned}\eta_1 &= 2\left(\frac{1+x}{1-y}\right) - 1 \\ \eta_2 &= y\end{aligned}$$

is called the collapsed-coordinate mapping is a singular transformation between the triangle and the biunit square. The set $\{D^{p,q}(x, y)\}_{p,q\geq 0}^{p+q\leq n}$ forms a basis for polynomoials of degree n. Moreover, they are orthogonal in the $L^2(K)$ inner product. Recently (**?**), it has been shown that these polynomials may be computed directly on the triangle without reference to the singular mapping. This means that no special treatment of the singular point is required, allowing use of standard automatic differentiation techniques to compute derivatives.

The recurrences are obtained by rewriting the polynomials as

$$D^{p,q}(x, y) = \chi^p(x, y)\psi^{p,q}(y),$$

where

$$\chi^p(x, y) = P_p^{0,0}(\eta_1) \left(\frac{1-\eta_2}{2}\right)^p$$

and

$$\psi^{p,q}(y) = P_q^{2p+1,0}(\eta_2) = P_q^{2p+1,0}(y).$$

This representation is not separable in η_1 and η_2, which may seem to be a drawback to readers familiar with the usage of these polynomials in spectral methods. However, they do still admit sum-factorization techniques. More importantly for present purposes, each χ^p is in fact a polynomial in x and y and may be computed by recurrence. $\psi^{p,q}$ is just a Jacobi polynomial in y and so has a well-known three-term recurrence. The recurrences derived in (**?**) are presented in Algorithm 5

Algorithm 5 Computes all triangular orthogonal polynomials up to degree d by recurrence

```
1: $D^{0,0}(x,y) := 1$
2: $D^{1,0}(x,y) := \frac{1+2x+y}{2}$
3: **for** $p \leftarrow 1,\, d-1$ **do**
4:     $D^{p+1,0}(x,y) := \left(\frac{2p+1}{p+1}\right)\left(\frac{1+2x+y}{2}\right) D^{p,0}(x,y) - \left(\frac{p}{p+1}\right)\left(\frac{1-y}{2}\right)^2 D^{p-1,0}(x,y)$
5: **end for**
6: **for** $p \leftarrow 0, d-1$ **do**
7:     $D^{p,1}(x,y) := D^{p,0}(x,y)\left(\frac{1+2p+(3+2p)y}{2}\right)$
8: **end for**
9: **for** $p \leftarrow 0, d-1$ **do**
10:     **for** $q \leftarrow 1, d-p-1$ **do**
11:         $D^{p,q+1}(x,y) := \left(a_q^{2p+1,0}y + b_q^{2p+1,0}\right) D^{p,q}(x,y) - c_q^{2p+1,0} D^{p,q-1}(x,y)$
12:     **end for**
13: **end for**
```

15.3 Representing polynomials and functionals

Even using recurrence relations and `numpy` vectorization for arithmetic, further care is required to optimize performance. In this section, standard operations on polynomials will be translated into vector operations, and then batches of such operations cast as matrix multiplication. This helps eliminate the interpretive overhead of Python while moving numerical computation into optimized library routines, since `numpy.dot` wraps level 3 BLAS and other functions such as `numpy.svd` wrap relevant LAPACK routines.

Since polynomials and functionals over polynomials both form vector spaces, it is natural to represent each of them as vectors representing expansion coefficients in some basis. So, let $\{\phi_i\}$ be the set of polynomials described above.

Now, any $p \in P$ is written as a linear combination of the basis functions $\{\phi_i\}$. Introduce a mapping $\mathcal{R}$ from P into $\mathbb{R}^{|P|}$ by taking the expansion coefficients of p in terms of $\{\phi_i\}$. That is,

$$p = \mathcal{R}(p)_i \phi_i,$$

where summation is implied over i.

A polynomial p may then be evaluated at a point x as follows. Let Φ_i be the basis functions tabulated at x. That is,

$$\Phi_i = \phi_i(x). \tag{15.1}$$

Then, evaluating p follows by a simple dot product:

$$p(x) = \mathcal{R}(p)_i \Phi_i. \tag{15.2}$$

More generally in FIAT, a set of polynomials $\{p_i\}$ will need to be evaulated simultaneously, such as evaluating all of the members of a finite element basis. The coefficients of the set of polynomials may be stored in the rows of a matrix C, so that

$$C_{ij} = \mathcal{R}(p_i)_j.$$

Tabulating this entire set of polynomials at a point x is simply obtained by matrix-vector multiplication. Let Φ_i be as in (15.1). Then,

$$p_i(x) = C_{ij}\Phi_j.$$

The basis functions are typically needed at a set of points, such as those of a quadrature rule. Let $\{x_j\}$ now be a collection of points in K and let

$$\Phi_{ij} = \phi_i(x_j),$$

where the rows of Φ run over the basis functions and the columns over the collection of points. As before, the set of polynomials may be tabulated at all the points by

$$p_i(x_j) = C_{ik}\Phi_{kj},$$

which is just the matrix product $C\Phi$ and may may be efficiently carried out by a library operation, such as the `numpy.dot` wrapper to level 3 BLAS.

Finite element computation also requires the evaluation of derivatives of polynomials. In a symbolic context, differentiation presents no particular difficulty, but working in a numerical context requires some special care.

For some differential operator D, the derivatives $D\phi_i$ are computed at apoint x, any polynomial $p = \mathcal{R}(p)_i\phi_i$ may be differentiated at x by

$$Dp(x) = \mathcal{R}(p)_i(D\phi_i),$$

which is exactly analagous to (15.2). By analogy, sets of polynomials may be differentiated at sets of points just like evaluation.

The formulae in Algorithm 5 and their tetrahedral counterpart are fairly easy to differentiate, but derivatives may also be obtained through automatic differentiation. Some experimental support for this using the AD tools in Scientific Python has been developed in an unreleased version of FIAT.

The released version of FIAT currently evaluates derivatives in terms of linear operators, which allows the coordinate singularity in the standard recurrence relations to be avoided. For each Cartesian partial derivative $\frac{\partial}{\partial x_k}$, a matrix D^k is calculated such that

$$\mathcal{R}\left(\frac{\partial p}{\partial x_k}\right)_i = D^k_{ij}\mathcal{R}(p)_j.$$

Then, derivatives of sets of polynomials may be tabulated by premultiplying the coefficient matrix C with such a D^k matrix.

This paradigm may also be extended to vector- and tensor-valued polynomials, making use of the multidimensional arrays implemented in numpy. Let P be a space of scalar-valued polynomials and $n > 0$ an integer. Then, a member of $(P)^m$, a vector with m components in P, may be represented as a two-dimensional array. Let $p \in (P)^m$ and p^i be the j^{th} component of p. Then $p = \mathcal{R}(p)_{jk}\phi_k$, so that $\mathcal{R}(p)_{jk}$ is the coefficient of ϕ_k for p^j.

The previous discussion of tabulating collections of functions at collections of points is naturally extended to this context. If $\{p_i\}$ is a set of members of P^m, then their coefficients may be stored in an array C_{ijk}, where C_i is the two-dimensional array $\mathcal{R}(p)_{jk}$ of coefficients for p_i. As before, $\Phi_{ij} = \phi_i(x_j)$ contains the of basis functions at a set of points. Then, the j^{th} component of v_i at the point x_k is naturally given by a three-dimensional array

$$p_i(x_k)^j = C_{ijl}\phi_{lk}.$$

Psychologically, this is just a matrix product if C_{ijl} is stored contiguously in generalized row-major format, and the operation is readily cast as dense matrix multiplication.

Returning for the moment to scalar-valued polynomials, linear functionals may also be represented as Euclidean vectors. Let $\ell : P \to \mathbb{R}$ be a linear functional. Then, for any $p \in P$,

$$\ell(p) = \ell(\mathcal{R}(p)_i\phi_i) = \mathcal{R}(p)_i\ell(\phi_i),$$

so that ℓ acting on p is determined entirely by its action on the basis $\{\phi_i\}$. As with $\mathcal{R}$, define $\mathcal{R}' : P' \to \mathbb{R}^{|P|}$ by

$$\mathcal{R}'(\ell)_i = \ell(\phi_i),$$

so that

$$\ell(p) = \mathcal{R}'(\ell)_i\mathcal{R}(p)_i.$$

Note that the inverse of $\mathcal{R}'$ is the Banach-space adjoint of $\mathcal{R}$.

Just as with evaluation, sets of linear functionals can be applied to sets of functions via matrix multiplication. Let $\{\ell_i\}_{i=1}^N \subset P'$ and $\{u_i\}_{i=1}^N \subset P$. The functionals are represented by a matrix

$$L_{ij} = \mathcal{R}'(\ell_i)_j$$

and the functions by

$$C_{ij} = \mathcal{R}(u_i)_j$$

Then, evaluating all of the functionals on all of the functions is computed by the matrix product

$$A_{ij} = L_{ik}C_{jk}, \tag{15.3}$$

or $A = LC^t$. This is especially useful in the setting of the next section, where the basis for the finite element spaceneeds to be expressed as a linear combination of orthogonal polynomials.

Also, the formalism of $\mathcal{R}'$ may be generalized to functionals over vector-valued spaces. As before, let P be a space of degree n with basis $\{\phi_i\}_{i=1}^{|P|}$ and to each $v \in (P)^m$ associate the representation $v^i = \mathcal{R}(v)_{ij}\phi_j$. In this notation, $v = \mathcal{R}(v)_{ij}\phi_j$ is the vector indexed over i. For any functional $\ell \in ((P)^m)'$, a representation $\mathcal{R}'(\ell)_{ij}$ must be defined such that

$$\ell(v) = \mathcal{R}'(\ell)_{ij}\mathcal{R}(v)_{ij},$$

with summation implied over i and j. To determine the representation of $\mathcal{R}'(f)$, let e^j be the canonical basis vector with $(e^j)_i = \delta_{ij}$ and write

$$\begin{aligned} \ell(v) &= \ell(\mathcal{R}_{ij}\phi_j) \\ &= \ell(\mathcal{R}(v)_{ij}\delta_{ik}e^k\phi^j) \\ &= \ell(\mathcal{R}(v)_{ij}e^i\phi_j) \\ &= \mathcal{R}(v)_{ij}\ell(e^i\phi_j). \end{aligned} \tag{15.4}$$

From this, it is seen that $\mathcal{R}'(\ell)_{ij} = \ell(e^i\phi_j)$.

Now, let $\{v_i\}_{i=1}^N$ be a set of vector-valued polynomials and $\{\ell_i\}_{i=1}^M$ a set of linear functionals acting on them. The polynomials may be stored by a coefficient tensor $C_{ijk} = \mathcal{R}(v_i)_{jk}$. The functionals may be represented by a tensor $L_{ijk} = \mathcal{R}'(\ell_i)_{jk}$. The matrix $A_{ij} = \ell_i(v_j)$ is readily computed by the contraction

$$A_{ij} = L_{ikl}C_{jkl}.$$

Despite having three indices, this calculation may still be performed by matrix multiplication. Since `numpy` stores arrays in row-major format, a simple reshaping may be performed without data motion so that $A = \tilde{L}\tilde{C}^t$, for $\tilde{L}$ and $\tilde{C}$ reshaped to two-dimensional arrays by combining the second and third axes.

15.4 Other polynomial spaces

Many of the complicated elements that motivate the development of a tool like FIAT polynomial spaces that are not polynomials of some complete degree (or vectors or tensors of such). Once a process for providing bases for such spaces is described, the techniques of the previous section may be applied directly. Most finite element polynomial spaces are described either by adding a few basis functions to some polynomials of complete degree or else by constraining such a space by some linear functionals.

15.4.1 Supplemented polynomial spaces

A classic example of the first case is the Raviart-Thomas element, where the function space of order r is

$$RT_r = (P_r(K))^d \oplus \left(\tilde{P}_r(K)\right) x,$$

where $x \in \mathbb{R}^d$ is the coordinate vector and $\tilde{P}_r$ is the space of homogeneous polynomials of degree r. Given any basis $\{\phi_i\}$ for $P_r(K)$ such as the Dubiner basis, it is easy to obtain a basis for $(P_r(K))^d$ by taking vectors where one component is some ϕ_i and the rest are zero. The issue is obtaining a basis for the entire space.

Consider the case $d = 2$ (triangles). While monomials of the form $x^i y^{r-i}$ span the space of homoegeneous polynomials, they are subject to ill-conditioning in numerical computations. On the other hand, the Dubiner basis of order r, $\{\phi_i\}_{i=1}^{|P_r|}$ may be ordered so that the last $r+1$ functions, $\{\phi_i\}_{i=|P_r|-r}^{|P_r|}$, have degree exactly r. While they do not span $\tilde{P}_r$, the span of $\{x\phi_i\}_{i=|P_r|-r}^{|P_r|}$ together with a basis for $(P_r(K))^2$ does span RT_r.

So, this gives a basis for the Raviart-Thomas space that can be evaluated and differentiated using the recurrence relations described above. A similar technique may be used to construct elements that consist of standard elements augmented with some kind of bubble function, such as the PEERS element of elasticity or MINI element for Stokes flow.

15.4.2 Constrained polynomial spaces

An example of the second case is the Brezzi-Douglas-Fortin-Marini element (**?**). Let $\mathcal{E}(K)$ be the set of dimension one cofacets of K (edges in 2d, faces in 3d). Then the function space is

$$BDFM_r(K) = \{u \in (P_r(K))^d : u \cdot n|_\gamma \in P_{r-1}(\gamma),\ \ \gamma \in \mathcal{E}(K)\}$$

This space is naturally interpreted as taking a function space, $(P_r(K))^d$, and imposing linear constraints. For the case $d = 2$, there are exactly three such constraints. For $\gamma \in \mathcal{E}(K)$, let μ^γ be the Legendre polynomial of degree r mapped to γ. Then, if a function $u \in (P_r(K))^d$, it is in $BDFM_r(K)$ iff

$$\int_\gamma (u \cdot n)\mu^\gamma \, ds = 0$$

for each $\gamma \in \mathcal{E}(K)$.

Number the edges by $\{\gamma_i\}_{i=1}^3$ and introduce linear functionals
$\ell_i(u) = \int_{\gamma_i} (u \cdot n)\mu^{\gamma_i} \, ds$. Then,

$$BDFM_r(K) = \cap_{i=1}^3 \text{null}(\ell_i).$$

This may naturally be cast into linear algebra and so evaluated with LAPACK. Following the techniques for constructing Vandermonde matrices, a *constraint matrix* may be constructed. Let $\{\bar{\phi}_i\}$ be a basis for $(P_r(K))^2$. Define the $3 \times |(P_r)|^2$ matrix

$$C_{ij} = \ell_i(\phi_j).$$

Then, a basis for the null space of this matrix is constructed using the singular value decomposition (**?**). The vectors of this null-space basis are readily seen to contain the expansion coefficients of a basis for $BDFM_r$ in terms of a basis for $P_r(K)^2$. With this basis in hand, the nodal basis for $BDFM_r(K)$ is obtained by constructing the generalized Vandermonde matrix.

This technique may be generalized to three dimensions, and it also applies to Nédélec (**?**), Arnold-Winther (**?**), Mardal-Tai-Winther (**?**), and many other elements.

15.5 Conveying topological information to clients

Most of this chapter has provided techniques for constructing finite element bases and evaluating and differentiating them. FIAT must also indicate which degrees of freedom are associated with which entities of the reference element. This information is required when local-global mappings are generated by a form compiler such as `ffc`.

The topological information is provided by a graded incidence relation and is similar to the presentation of finite element meshes in (**?**). Each entity in the reference element is labeled by its topological dimension (e.g. 0 for vertices and 1 for edges), and then the entities of the same dimension are ordered by some convention. To each entity, a list of the local nodes is associated. For example, the reference triangle with entities labeled is shown in Figure 15.5, and the cubic Lagrange triangle with nodes in the dual basis labeled is shown in Figure 15.5.

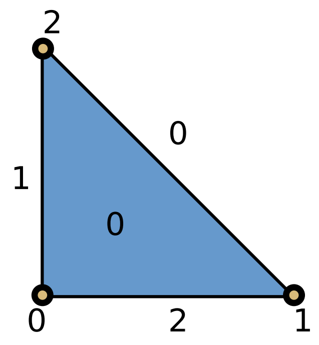


Figure 15.1: The reference triangle, with vertices, edges, and the face numbered.

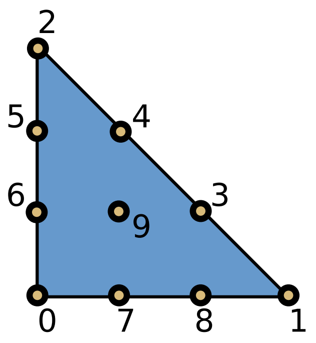


Figure 15.2: The cubic Lagrange triangle, with nodes in the dual basis labelled.

For this example, the graded incidence relation is stored as

```
{ 0: { 0: [ 0 ] ,
       1: [ 1 ] ,
       2: [ 2 ] } ,
  1: { 0: [ 3 , 4 ] ,
       1: [ 5 , 6 ] ,
       2: [ 7 , 8 ] } ,
  2: { 0: [ 9 ] } }
```

15.6 Functional evaluation

In order to construct nodal interpolants or strongly enforce boundary conditions, FIAT also provides information to numerically evaluate linear functionals. These rules are typically exact for a certain degree polynomial and only approximate on general functions. For scalar functions, these rules may be represented by a collection of points and corresponding weights $\{x_i\}, \{w_i\}$ so that

$$\ell(f) \approx w_i f(x_i).$$

For example, pointwise evaluation at a point x is simply represented by the coordinates of x together with a weight of 1. If the functional is an integral moment, such as

$$\ell(f) = \int_K gf \, dx,$$

then the points $\{x_i\}$ will be those of some quadrature rule and the weights will be $w_i = \omega_i g(x_i)$, where the ω_i are the quadrature weights.

This framework is extended to support vector- and tensor-valued function spaces, by including a component corresponding to each point and weight. If v is a vector-valued function and v_α is its component, then functionals are written in the form

$$\ell(v) \approx w_i v_{\alpha_i}(x_i),$$

so that the sets of weights, components, and points must be conveyed to the client.

This framework may also support derivative-based degrees of freedom by including a multiindex at each point corresponding to a particular partial derivative.

15.7 Overview of fundamental class structure

Many FEniCS users will never directly use FIAT; for them, interaction will be moderated through a form compiler such as `ffc`. Others will want to use the FIAT basis functions in other contexts. At a basic level, a user will access FIAT through top-level classes such as `Lagrange` and `RaviartThomas` that implement the elements. Typically, the class constructors accept the reference element and order of function space as arguments. This gives an interface that is parametrized by dimension and degree. The classes such as `Lagrange` derive from a base class `FiniteElement` that provides access to the three components of the Ciarlet triple.

The currently released version of FIAT stores the reference element as a flag indicating the simplex dimension, although a development version provides an actual class describing reference element geometry and topology. This will allow future releases of FIAT to be parametrized over the particular reference element shape and topology.

The function space P is modelled by the base class `PolynomialSet`, while contains a rule for constructing the base polynomials ϕ_i (e.g. the Dubiner basis) and a multidimensional array of expansion coefficients for the basis of P. Special subclasses of this provide (possibly array-valued) orthogonal bases as well as the rules for constructing supplemented and constrained bases. These classes provide mechanisms for tabulating and differentiating the polynomials at input points as well as basic queries such as the dimension of the space.

The set of finite element nodes is similarly modeled by a class `DualBasis`. This provides the functionals of the dual basis as well as their connection to the reference element facets. The functionals are modeled by a `FunctionalSet` object, which is a collection of `Functional` objects. Each `Functional` object contains a reference to the `PolynomialSet` over which it is defined and the array of coefficients representing it and owns a `FunctionalType` class providing the information described in the previous section. The `FunctionalSet` class batches these coefficients together in a single large array.

The constructor for the `FiniteElement` class takes a `PolynomialSet` modeling the starting basis and a `DualBasis` defined over this basis and constructs a new `PolynomialSet` by building and inverting the generalized Vandermonde matrix.

Beyond this basic finite element structure, FIAT provides quadrature such as Gauss-Jacobi rules in one dimension and collapsed-coordinate rules in higher dimensions. It also provides routines for constructing lattices of points on eah of the reference element shapes and their facets.

In the future, FIAT will include the developments discussed already (more general reference element geometry/topology and automatic differentiation). Automatic differentiation will make it easier to construct finite elements with derivative-type degrees of freedom such as Hermite, Morley, and Argyris. Aditionally, we hope to expand the collection of quadrature rules and provide more advanced point distributions, such as Warburton's warp-blend points (**?**).

CHAPTER 16

Instant: Just-in-Time Compilation of C/C++ Code in Python

By Ilmar M. Wilbers, Kent-Andre Mardal and Martin S. Alnæs

Chapter ref: **[wilbers]**

16.1 Introduction

Instant is a small Python module for just-in-time compilation (or inlining) of C/C++ code based on SWIG (SWIG software package) and Distutils[1]. Just-in-time compilation can significantly speed up, e.g., your NumPy (Numerical Python software packa code in a clean and readable way. This makes Instant a very convenient tool in combination with code generation. Before we demonstrate the use of Instant in a series of examples, we briefly step through the basic ideas behind the implementation. Instant relies on SWIG for the generation of wrapper code needed for making the C/C++ code usable from Python (Python programming language). SWIG is a mature and well-documented tool for wrapping C/C++ code in many languages. We refer to its website for a comprehensive user manual and we also discuss some common tricks and troubles in Chapter **??**. The code to be inlined, in addition to the wrapper code, is then compiled into a Python extension module (a shared library with functionality as specified by the Python C-API) by using Distutils. To check whether the C/C++ code has changed since the last execution, Instant computes the SHA1 sum[2] of the code and compares it to the SHA1 checksum of the code used in the previous execution. Finally, Instant has implemented

[1]http://www.python.org/doc/2.5.2/lib/module-distutils.html

[2]http://www.apps.ietf.org/rfc/rfc3174.html

a set of SWIG typemaps, allowing the user to transfer NumPy arrays between the Python code and the C/C++ code.

There exist several packages that are similar to Instant. Worth mentioning here are Weave (Weave: Tools for inlining C/C++ in Python), Cython (Cython: C-Extensions for P
and F2PY (Peterson). Weave allows us to inline C code directly in our Python code. Unlike Instant, Weave does not require the specification of the function signature and the return argument. For specific examples of Weave and the other mentioned packages, we refer to (I. M. Wilbers and H. P. Langtangen and Å. Ødegård, 2009, Weave: Tools for inlining C/C++ in Python). Weave is part of SciPy (SciPy software packag
F2PY is currently part of NumPy, and is primarily intended for wrapping Fortran code. F2PY can also be used for wrapping C code. Cython is a rather new project, branched from the more well-known Pyrex project (Pyrex – a Language for Writing Python Exte
Cython is attractive because of its integration with NumPy arrays. Cython differs from the other projects by being a programming language of its own. Cython extends Python with concepts such as static typing, hence allowing the user to incrementally speed up the code.

Instant accepts plain C/C++. This makes it particularly attractive to combine Instant with tools capable of generating C/C++ code such as FFC (see Chapter 13), SFC (see Chapter 17), Swiginac (Swiginac Python interface to GiNaC), and Sympy (Certik et al., 2009). In fact, tools like these have been the main motivation behind Instant, and both FFC and SFC employ Instant. Instant is released under a BSD license, see the file `LICENSE` in the source directory.

In this chapter we will begin with several examples in Section 16.2. Section 16.3 explains how Instant works, while Section 16.4 gives a detailed description of the API.

16.2 Examples

All code from the examples in this section can be found online[3]. We will refer to this location as `$examples`.

16.2.1 Installing Instant

Before trying to run the examples, you need to install Instant. The latest Instant release can be downloaded from the FEniCS website (FEniCS). It is available both as a source code tarball and as a Debian package. In addition, the latest source code can be checked out using Mercurial (Mercurial software package):

```
hg clone http://www.fenics.org/hg/instant
```

[3]http://www.fenics.org/pub/documents/book/instant/examples

Installing Instant from the source code is done with a regular Distutils script, i.e,

```
python setup.py install
```

After successfully installing Instant, one can verify the installation by running the scripts `run_tests.py` followed by `rerun_tests.py` in the `tests`-directory of the source code. The first will run all the examples after having cleaned the Instant cache, the second will run all examples using the compiled modules found in the Instant cache from the previous execution.

16.2.2 Hello World

Our first example demonstrate the usage of Instant in a very simple case:

```
from instant import inline
c_code = r'''
double add(double a, double b)
{
  printf("Hello world! C function add is being called...\n");
  return a+b;
}'''
add_func = inline(c_code)
sum = add_func(3, 4.5)
print 'The sum of 3 and 4.5 is', sum
```

Here Instant will wrap the C-function `add` into a Python extension module by using SWIG and Distutils. The inlined function is written in standard C. SWIG supports almost all of C and C++, including object orientation and templates. When running this Python snippet the first time, compiling the C code takes a few seconds. Next time we run it, however, the compilation is omitted, given that no changes are made to the C source code.

Note that a raw string is used in this example, to avoid Python interpreting escape sequences such as '\n'. Alternatively, special characters can be escaped using a backslash.

Although Instant notifies the user when it is compiling, it might sometimes be necessary, e.g. when debugging, to see the details of the Instant internals. We can do this by setting the logging level before calling any other Instant functions:

```
from instant import output
output.set_logging_level('DEBUG')
```

The intrinsic Python module `logging` is used. First, the `build_function` arguments are displayed, whereafter the different steps performed by Instant are shown in detail, e.g whether the module is found in cache and the arguments to the Distutils file when building the module. This example can be found in the file `$examples/ex1.py`.

16.2.3 NumPy Arrays

One basic problem with wrapping C and C++ code is how to handle dynamically allocated arrays. Arrays allocated dynamically are typically represented in C/C++ by a pointer to the first element of an array and a separate integer variable holding the array size. In Python the array variable is itself an object contains the data array, array size, type information etc. However, a pointer in C/C++ does not necessarily represent an array. Therefore, SWIG provides the typemap functionality that allows the user to specify a mapping between Python and C/C++ types. We will not go into details on typemaps in this chapter, but the reader should be aware that it is a powerful tool that may greatly enhance your code, but also lead to mysterious bugs when used wrongly. Typemaps are discussed in Chapter **??** and at length at the SWIG webpage. In this chapter, it is sufficient to illustrate how to deal with arrays in Instant using the NumPy module. More details on how Instant NumPy arrays can be found in Section 16.3.1.

16.2.4 Ordinary Differential Equations

We introduce a solver for an ordinary differential equation (ODE) modeling blood pressure by using a Windkessel model. The ODE is as follows:

$$\frac{d}{dt}p(t) = BQ(t) - Ap(t), \quad t \in (0,1), \tag{16.1}$$

$$p(0) = p0. \tag{16.2}$$

Here $p(t)$ is the blood pressure, $Q(t)$ is the volume flux of blood, A is ... and B is .... An explicit scheme is:

$$p_i = p_{i-1} + \Delta t(BQ_i - Ap_{i-1}), \quad \text{for} \quad i = 1, \ldots, N-1, \tag{16.3}$$

$$p_0 = p0. \tag{16.4}$$

The scheme can be implemented in Python as follows using NumPy arrays:

```
def time_loop_py(p, Q, A, B, dt, N, p0):
    p[0] = p0
    for i in range(1, N):
        p[i] = p[i-1] + dt*(B*Q[i] - A*p[i-1])
```

The C code given as argument to the Instant function `inline_with_numpy` looks like:

```
void time_loop_c(int n, double* p,
                 int m, double* Q,
                 double A, double B,
                 double dt, int N, double p0)
{
    if ( n != m || N != m )
    {
        printf("n, m and N should be equal\n");
        return;
    }

    p[0] = p0;
    for (int i=1; i<n; i++)
    {
        p[i] = p[i-1] + dt*(B*Q[i] - A*p[i-1]);
    }
}
```

In this example, `(int n, double* p)` represents an array of doubles with length n. However, this can not be determined by the function signature:

```
void time_loop_C(int n, double* p, int m, double* Q, ...)
```

For example, `double* p` may be an array of length m or it may simply be output. In Instant you can specify 1-dimensional arrays as follows:

```
time_loop_c = inline_with_numpy(c_code,
                                arrays = [['n', 'p'],
                                          ['m', 'Q']])
```

Here we tell Instant that `(int n, double* p)` and `(int m, double* Q)` are NumPy arrays (and Instant then employs a few typemaps). We may then call the `time_loop` function as follows:

```
time_loop_c(p, Q, 1.0, 1.0, 1.0/(N-1), N, 1.0)
```

In the above example we obtain a speed-up of about a factor 400 when using 100000 time steps compared to the pure Python with NumPy version, see Table 16.1. This is about the same as a pure C program. The result of solving the ODE can be seen in Figure 16.1. The comparison between NumPy and Instant is

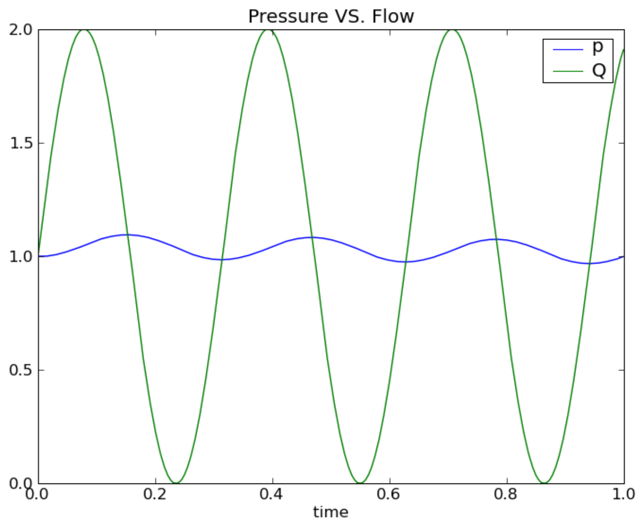


Figure 16.1: Plot of pressure and blood volume flux computed by solving the Windkessel model.

not really fair, as NumPy primarily gives a speed-up for code that can be vectorized, something that is not the case with our current ODE. In fact, utilizing pure Python lists instead of NumPy arrays, reduces the speed-up to a factor 100. For code that can be vectorized, the speed-up is about one order of magnitude when we use Instant instead (I. M. Wilbers and H. P. Langtangen and Å. Ødegård, 2009).

N	100	1000	10000	100000	1000000
CPU time with NumPy	3.9e-4	3.9e-3	3.8e-2	3.8e-1	3.8
CPU time with Python	0.7e-4	0.7e-3	0.7e-2	0.7e-1	0.7
CPU time with Instant	5.0e-6	1.4e-5	1.0e-4	1.0e-3	1.1e-2
CPU time with C	4.0e-6	1.1e-5	1.0e-4	1.0e-3	1.1e-2

Table 16.1: CPU times of Windkessel model for different implementations (in seconds).

The complete code for this example can be found in `$examples/ex2.py`

16.2.5 Numpy Arrays and OpenMP

It is easy to speed up code on parallel computers with OpenMP. We will not describe OpenMP in any detail here, the reader is referred to (OpenMP Application Program Interf
However, note that preprocessor directives like '`#pragma omp ...`' are OpenMP directives and that OpenMP functions start with `omp`. In this example, we want

to solve a standard 2-dimensional wave equation in a heterogeneous medium with local wave velocity k:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [k \nabla u] \,. \tag{16.5}$$

We set the boundary condition to $u = 0$ for the whole boundary of a rectangular domain $\Omega = (0,1) \times (0,1)$. Further, u has the initial value $I(x,y)$ at $t = 0$ while $\partial u/\partial t = 0$. We solve the wave equation using the following finite difference scheme:

$$\begin{aligned} u^l_{i,j} = & \left(\frac{\Delta t}{\Delta x}\right)^2 [k_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - k_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})]^{l-1} \\ & + \left(\frac{\Delta t}{\Delta y}\right)^2 [k_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - k_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1})]^{l-1}. \end{aligned} \tag{16.6}$$

Here, $u^l_{i,j}$ represents u at the grid point x_i and y_j at time level t_l, where

$$\begin{aligned} & x_i = i\Delta x, i = 0, \ldots, n \\ & y_i = j\Delta y, j = 0, \ldots, m \text{ and} \\ & t_l = l\Delta t, \end{aligned}$$

Also, $k_{i+\frac{1}{2},j}$ is short for $k(x_{i+\frac{1}{2}}, y_j)$.

The code for calculating the next time step using OpenMP looks like:

```
void stencil(double dt, double dx, double dy,
             int ux, int uy, double* u,
             int umx, int umy, double* um,
             int kx, int ky, double* k,
             int upn, double* up){
#define index(u, i, j) u[(i)*m + (j)]
  int i=0, j=0, m = ux, n = uy;
  double hx, hy, k_c, k_ip, k_im, k_jp, k_jm;
  hx = pow(dt/dx, 2);
  hy = pow(dt/dy, 2);
  j = 0;   for (i=0; i<m; i++) index(up, i, j) = 0;
  j = n-1; for (i=0; i<m; i++) index(up, i, j) = 0;
  i = 0;   for (j=0; j<n; j++) index(up, i, j) = 0;
  i = m-1; for (j=0; j<n; j++) index(up, i, j) = 0;
  #pragma omp for
  for (i=1; i<m-1; i++){
    for (j=1; j<n-1; j++){
      k_c = index(k, i, j);
      k_ip = 0.5*(k_c + index(k, i+1, j));
      k_im = 0.5*(k_c + index(k, i-1, j));
      k_jp = 0.5*(k_c + index(k, i, j+1));
      k_jm = 0.5*(k_c + index(k, i, j-1));
      index(up, i, j) = 2*index(u, i, j) - index(um, i, j) +
        hx*(k_ip*(index(u, i+1, j) - index(u, i, j)) -
             k_im*(index(u, i, j) - index(u, i-1, j))) +
        hy*(k_jp*(index(u, i, j+1) - index(u, i, j)) -
            k_jm*(index(u, i, j) - index(u, i, j-1)));
    }
  }
}
```

We also need to add the OpenMP header `omp.h` and compile with the flag `-fopenmp` and link with the OpenMP shared library, e.g. `libgomp.so` for Linux (specified with `-lgomp`). This can be done as follows:

```
instant_ext = \
  build_module(code=c_code,
               system_headers=['numpy/arrayobject.h',
                               'omp.h'],
               include_dirs=[numpy.get_include()],
               init_code='import_array();',
               cppargs=['-fopenmp'],
               lddargs=['-lgomp'],
               arrays=[['ux', 'uy', 'u'],
               ['umx', 'umy', 'um'],
               ['kx', 'ky', 'k'],
               ['upn', 'up', 'out']])
```

Note that the arguments `include_headers`, `init_code`, and the first element of `system_headers` could have been omitted had we chosen to use `inline_module_with_numpy` instead of `build_module`. We could also have used `inline_with_numpy`, which would have returned only the function, not the whole module. For more details, see the next section. The complete code can be found in `$examples/ex3.py`. It might very well be possible to write more efficient code for many of these examples, but the primary objective is to examplify different Instant features.

16.3 Instant Explained

The previous section concentrated on the usage of Instant and it may appear mysterious how it actually works since it is unclear what files that are made during execution and where they are located. In this section we explain this.

We will again use our first example, but this time with the keyword argument `modulename` set explicitely. The file can be found under `$examples/ex4.py`:

```
from instant import inline
code = r'''
double add(double a, double b)
{
  printf("Hello world! C function add is being called...\n");
  return a+b;
}'''
add_func = inline(code, modulename='ex4_cache')
sum = add_func(3, 4.5)
print 'The sum of 3 and 4.5 is', sum
```

Upon calling Instant the first time for some C/C++ code, Instant compiles this code and stores the resulting files in a directory `ex4_cache`. The output from running the code the first time is:

```
--- Instant: compiling ---
Hello world! C function add is being called...
The sum of 3 and 4.5 is 7.5
```

Next time we ask Instant to call this code, it will check if the compiled files are available either in cache or locally, and further whether we need to rebuild these files based on the checksum of the source files and the arguments to the Instant function. This means that Instant will perform the compile step *only* if changes are made to the source code or arguments. More details about the different caching options can be found in Section 16.3.2.

The resulting module files can be found in a directory reflecting the name of the module, in this case `ex4_cache`:

```
ilmarw@multiboot:~/instant_doc/code$ cd ex4_cache/
ilmarw@multiboot:~/instant_doc/code/ex4_cache$ ls -g
total 224
drwxr-xr-x 4 ilmarw  4096 2009-05-18 16:52 build
-rw-r--r-- 1 ilmarw   844 2009-05-18 16:52 compile.log
-rw-r--r-- 1 ilmarw   183 2009-05-18 16:52 ex4_cache-0.0.0.egg-info
-rw-r--r-- 1 ilmarw    40 2009-05-18 16:52 ex4_cache.checksum
-rw-r--r-- 1 ilmarw   402 2009-05-18 16:53 ex4_cache.i
-rw-r--r-- 1 ilmarw  1866 2009-05-18 16:52 ex4_cache.py
-rw-r--r-- 1 ilmarw  2669 2009-05-18 16:52 ex4_cache.pyc
-rwxr-xr-x 1 ilmarw 82066 2009-05-18 16:52 _ex4_cache.so
-rw-r--r-- 1 ilmarw 94700 2009-05-18 16:52 ex4_cache_wrap.cxx
-rw-r--r-- 1 ilmarw    23 2009-05-18 16:53 __init__.py
-rw-r--r-- 1 ilmarw   448 2009-05-18 16:53 setup.py
```

When building a new module, Instant creates a new directory with a number of files. The first file it generates is the SWIG interface file, named `ex4_cache.i` in this example. Then the Distutils file `setup.py` is generated based and executed. During execution, `setup.py` first runs SWIG in the interface file, producing `ex4_cache_wrap.cxx` and `ex4_cache.py`. The first file is then compiled into a shared library `_ex4_cache.so` (note the leading underscore). A file `ex4_cache-0.0.0.egg-info` and a directory `build` will also be present as a result of these steps. The output from executing the Distutils file is stored in the file `compile.log`. Finally, a checksum file named `ex4_cache.checksum` is generated, containing a checksum based on the files present in the directory. The final step consists of moving the whole directory from its temporary location to either cache or a user-specified directory. The `__init__.py` imports the module `ex4_cache`.

The script `instant-clean` removes compiled modules from the Instant cache, located in the directory `.instant` in the home directory of the user running it. In addition, all Instant modules located in the temporary directory where they were first generated and compiled. It does not clean modules located elsewhere.

The script `instant-showcache` allow you to see the modules currently located in the Instant cache:

```
Found 1 modules in Instant cache:
test_cache
Found 1 lock files in Instant cache:
test_cache.lock
```

Arguments to this script will output the files matching the specified pattern, for example will `instant-showcache 'test*.i'` show the content of the SWIG interface file for any module beginning with the letters `test`.

16.3.1 Arrays and Typemaps

Instant has support for converting NumPy arrays to C arrays and vice versa. For arrays with up to three dimensions, the SWIG interface file from NumPy is used, with a few modifications. When installing Instant, this file is included as well. `arrays` should be a list, each entry containing information about a specific array. This entry should contain a list with strings, so the `arrays` argument is a nested list.

Each array (i.e. each element in `arrays`) is a list containing the names of the variables describing that array in the C code. For a 1-dimensional array, this means the names of the variables containing the length of the array (an `int`), and the array pointer (can have several tpes, but the default is `double`). For 2-dimensional arrays we need three strings, two for the length in each dimension, and the third for the array pointer. For 3-dimensional arrays, there will be three variables first. This example should make things clearer

```
arrays = [['len', 'a'],
          ['len_bx', 'len_by', 'b'],
          ['len_cx', 'len_cy', 'len_cz', 'c']]
```

These variables names specified reflect the variable names in the C function signature. It is important that the order of the variables in the signature is retained for each array, e.g. one cannot write:

```
c_code = """
double sum (int len_a, int len_bx, int len_by,
            double* a, double* b)
{
  ...
}
"""
```

The correct code would be:

```
c_code = """
double sum (int len_a, double* a,
            int len_bx,
            int len_by, double* b)
{
  ...
}
"""
```

The order of the arrays can be changed, as long as the arguments in the Python function are changed as well accordingly.

Data Types

Default, all arrays are assumed to be of type `double`, but several other types are supported. These are `float`, `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`. The type can be specified by adding an additional element to the list describing the array, e.g.

```
arrays = [['len', 'a', 'long']]
```

It is important that there is correspondance between the type of the NumPy array and the type in the signature of the C function. For arrays that are changed in-place, the types have to match exactly. For arrays that are input or output (see next section), one has to make sure that the implicit casting is done to a type with higher accuracy. For input arrays, the C type must be of higher (or the same) accuracy than the NumPy array, while for output arrays the NumPy array type must be of higher (or the same) accuracy than the C array. The NumPy type `float32` corresponds to the C type `float`, while `float64` corresponds to `double`. The NumPy type `float` is the same as `float64`. For integer arrays, the mapping between NumPy types and C types depends on your system. Using `long` as the C type will work in most cases.

Input/Output Arrays

All arrays are assumed to be both input and output arrays, i.e. any changes to arrays in the C code result in the NumPy array being changed in-place. For performace purposes, this is desirable, as we avoid unecessary copying of data. The NumPy SWIG interface file has support for both input and output arrays in addition to changing arrays in-place. Input arrays do not need to be NumPy arrays, but can be any type of sequence, e.g. lists and tuples. The default behaviour of the NumPy SWIG interface file is to create new objects for sequences that are not NumPy arrays, while using mere pointers to the data of NumPy arrays. Instant deviates from this behaviour by taking copies of all input data, allowing for the modification of the array in the C code, as might be necessary for certain applications, while retaining the array as seen from the Python code. An array is marked as input only by adding the additional element `'in'` to the list describing the array:

```
arrays = [['len', 'a', 'in']]
```

It is also possible to create output arrays in the C code. Instead of creating an array in the Python code and sending it as an in-place array to the C code, the array is created by the wrapper code and returned. If there are are multiple output arrays or the C function has a return argument, the wrapper function returns a tuple with the different arguments. This approach is more Python-like than changing arrays in-place.

We only need to specify the length of the array when calling the wrapper function. The limitation is that only 1-dimensional arrays are supported, which means that we need to set the shape of the array manually after calling the wrapper function. In the C code all arrays are treated as 1-dimensional, so this does not affect the C code. An array is marked as input only by adding the additional element `'out'` to the list describing the array. The following code shows an example where we calculate matrix-vector multiplication $x = Ab$. The matrix A is marked as input, the vector b as in-place, and the vector x as output. The example is only meant for illustrating the use of the different array options, and can be found in the file `$examples/ex5.py`. We verify that the result is correct by using the dot product from NumPy:

```
from instant import inline_with_numpy
from numpy import arange, dot

c_code = '''
void dot_c(int Am, int An, long* A, int bn, int* b,
           int xn, double* x)
{
  for (int i=0; i<Am; i++)
  {
    x[i] = 0;
    for (int j=0; j<An; j++)
    {
      x[i] += A[i*Am + j]*b[j];
    }
  }
}
'''
dot_c = \
  inline_with_numpy(c_code,
                    arrays = [['Am', 'An', 'A', 'in', 'long'],
                              ['bn', 'b', 'int'],
                              ['xn', 'x', 'out']])
a = arange(9)
a.shape = (3, 3)
b = arange(3)

c1 = dot_c(a, b, a.shape[1])
c2 = dot(a, b)
print c1
print c2
```

Multi-dimensional Arrays

If one needs to work with arrays that are more than 3-dimensional, this is possible. However, the typemaps used for this employ less error checking, and can only be used for the C type `double`. The list describing the array should contain the variable name for holding the number of dimensions, the variable name for an integer arrays holding the size in each dimension, the variable name for the array, and the argument `'multi'`, indicating that it has more than 3 dimensions. The `arrays` argument could for example be:

```
arrays = [['m', 'mp', 'ar1', 'multi'],
          ['n', 'np', 'ar2', 'multi']]
```

In this case, the C function signature should look like:

```
void sum (int m, int* mp, double* ar1, int n,
          int* np, double* ar2)
```

In the C code, all arrays are 1-dimensional. Indexing a 3-dimensional arrays becames rather complicated because of striding. For instance, instead of writing `u(i,j,k)` we need to write `u[i*ny*nz + j*ny + k]`, where `nx`, `ny`, and `nz` are the lengths of the array in each direction. One way of achieving a simpler syntax is to use the `#define` macro in C:

```
#define index(u, i, j, k) u[(i)*nz*ny + (j)*ny + (k)]
```

which allows us to write `index(u, i, j, k)` instead.

16.3.2 Module name, signature, and cache

The Instant cache resides in the directory `.instant` in the directory of the user. It is possible to specify a different directory, but the `instant-clean` script will not remove these when executed. The three keyword arguments `modulename`, `signature`, and `cache_dir` are connected. If none of them are given, the default behaviour is to create a signature from the contents of the files and arguments to the `build_module` function, resulting in a name starting with `instant_module_` followed by a long checksum. The resulting code is copied to Instant cache unless `cache_dir` is set to a specific directory. Note that changing the arguments or any of the files will result in a new directory in the Instant cache, as the checksum no longer is the same. Before compiling a module, Instant will always check if it is cached in both the Instant cache and in the current working directory.

If `modulename` is used, the directory with the resulting code is named accordingly, but not copied to the Instant cache. Instead, it is stored in the current working directory. Any changes to the argument or the source files will automatically result in a recompilation. The argument `cache_dir` is ignored.

When `signature` is given as argument, Instant uses this instead of calculating checksums. The resulting directory has the same name as the signature, provided the signature does not contain more than 100 characters containing only letters, numbers, or a underscore. If the signature contains any of these characters, the module name is generated based on the checksum of this string, resulting in a module name starting with `instant_module_` followed by the checksum. Because the user specifies the signature herself, changes in the arguments or source code will not cause a recompilation. The use of signatures is primarily intended for external software making use of Instant, e.g. SFC. Sometimes, the code output by this software might be different from the code used previously by Instant, without these changes affecting the result of running this code (e.g. comments are inserted to the code). By using signatures, the external program can decide when recompilation is necessary instead of leaving this to Instant. Unless otherwise specified, the modules is stored in the Instant cache.

It is not possible to specify both the module name and the signature. If both are given, Instant will issue an error.

In addition to the disk cache discussed so far, Instant also has a memory cache. All modules used during the life-time of a program are stored in memory for faster access. The memory cache is always checked before the disk cache.

16.3.3 Locking

Instant provides file locking functionality for cache modules. If multiple processes are working on the same module, race conditions could potentially occur whre two or more processes believe the module is missing from the cache and try to write it simultaneously. To avoid race conditions, lock files were introduced. The lock files reside in the Instant cache, and locking is only enabled for modules that should be cached, i.e. where the module name is not given explicitely as argument to `build_module` or one of its wrapper functions. The first process to reach the stage where the module is copied from its temporary location to the Instant cache, will aquire a lock, and other processes cannot access this module while it is being copied.

16.4 Instant API

In this section we will describe the various Instant functions and their arguments visible to the user. The first ten functions are the core Instant functions, with `build_module` being the main one, while the next eight are wrapper functions around this function. Further, there are four more helper functions available, intended for using Instant with other applications.

16.4.1 `build_module`

This function is the most important one in Instant, and for most applications the only one that developers need to use, combined with the existing wrapper functions around this function. The return argument is the compiled module, hence it can be used directly in the calling code (rather then importing it as a Python module). It is also possible to import the module manually if compiled in the same directory as the calling code.

There are a number of keyword arguments, and we will explain them in detail here. Although one of the aims of Instant is to minimize the direct interaction with SWIG, some of the keywords require a good knowledge of SWIG in order to make sense. In this way, Instant can be used both by programmers new to the use of extension languages for Python, as well as by experienced SWIG programmers. The keywords arguments are as follows:

- `modulename`
  - Default: `None`
  - Type: String
  - Comment: The name you want for the module. If specified, the module will not be cached. If missing, a name will be constructed based on a checksum of the other arguments, and the module will be placed in the global cache. See Section 16.3.2 for more details.
- `source_directory`
  - Default: '.'
  - Type: String
  - Comment: The directory where user supplied files reside. The files given in `sources`, `wrap_headers`, and `local_headers` are expected to exist in this directory.
- `code`
  - Default: `''`
  - Type: String
  - Comment: The C or C++ code to be compiled and wrapped.
- `init_code`
  - Default: `''`
  - Type: String
  - Comment: Code that should be executed when the Instant module is imported. This code is inserted in the SWIG interface file, and is used for instance for calling `import_array()` used for the initialization of NumPy arrays.
- `additional_definitions`
  - Default: `''`
  - Type: String

  - Comment: Additional definitions (typically needed for inheritance) for interface file. These definitions should be given as triple-quoted strings in the case they span multiple lines, and are placed both in the initial block for C/C++ code (`%{,%}`-block), and the main section of the interface file.

- `additional_declarations`
  - Default: `''`
  - Type: String
  - Comment: Additional declarations (typically needed for inheritance) for interface file. These declarations should be given as triple-quoted strings in the case they span multiple lines, and are placed in the main section of the interface file.

- `sources`
  - Default: `[]`
  - Type: List of strings
  - Comment: Source files to compile and link with the module. These files are compiled togehter with the SWIG-generated wrapper file into the final library file. Should reside in directory specified in `source_directory`.

- `wrap_headers`
  - Default: `[]`
  - Type: List of strings
  - Comment: Local header files that should be wrapped by SWIG. The files specified will be included both in the initial block for C/C++ code (with a C directive) and in the main section of the interface file (with a SWIG directive). Should reside in directory specified in `source_directory`.

- `local_headers`
  - Default: `[]`
  - Type: List of strings
  - Comment: Local header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive). Should reside in directory specified in `source_directory`.

- `system_headers`
  - Default: `[]`
  - Type: List of strings
  - Comment: System header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive).
- `include_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to search for header files for building the extension module. Needs to be absolute path names.
- `library_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to search for libraries (`-l`) for building the extension module. Needs to be absolute paths.
- `libraries`
  - Default: `[]`
  - Type: List of strings
  - Comment: Libraries needed by the Instant module. The libraries will be linked in from the shared object file. The initial `-l` is added automatically.
- `swigargs`
  - Default: `['-c++', '-fcompact', '-O', '-I.', '-small']`
  - Type: List of strings
  - Comment: Arguments to swig, e.g. `['-lpointers.i']` to include the SWIG pointers.i library.
- `swig_include_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to include in the 'swig' command.

- `cppargs`
  - Default: `['-O2']`
  - Type: List of strings
  - Comment: Arguments to the C++ compiler, other than include directories, e.g. `['-Wall', '-fopenmp']`.
- `lddargs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Arguments to the linker, other than libraries and library directories, e.g. `['-E', '-U']`.
- `arrays`
  - Default: `[]`
  - Type: List of strings
  - Comment: A nested list describing the C arrays to be made from NumPy arrays. The SWIG interface for fil NumPy is used. For 1D arrays, the inner list should contain strings with the variable names for the length of the arrays and the array itself. 2D matrices should contain the names of the dimensions in the two directions as well as the name of the array, and 3D tensors should contain the names of the dimensions in the three directions in addition to the name of the array. If the NumPy array har more than four dimensions, the inner list should contain strings with variable names for the number of dimensions, the length in each dimension as a pointer, and the array itself, respectively. For more details, see Section 16.3.1.
- `generate_interface`
  - Default: `True`
  - Type: Boolean
  - Comment: Indicate whether you want to generate the interface files.
- `generate_setup`
  - Default: `True`
  - Type: Boolean
  - Comment: Indicate if you want to generate the setup.py file.
- `signature`

  - Default: `None`
  - Type: String
  - Comment: A signature string to identify the form instead of the source code. See Section 16.3.2.

- `cache_dir`
  - Default: None
  - Type: String
  - Comment: A directory to look for cached modules and place new ones. If missing, a default directory is used. Note that the module will not be cached if `modulename` is specified. The cache directory should not be used for anything else.

16.4.2 `inline`

The function `inline` creates a module given that the input is a valid C/C++ function. It is only possible to inline one C/C++ function each time. One mandatory argument, which is the C/C++ code to be compiled.

The default keyword arguments from `build_module` are used, with `c_code` as the C/C++ code given as argument to `inline`. These keyword argument can be overridden, however, by giving them as arguments to `inline`, with the obvious exception of `code`. The function tries to return the single C/C++ function to be compiled rather than the whole module, if it fails, the module is returned.

16.4.3 `inline_module`

The same as `inline`, but returns the whole module rather than a single function. Except for the C/C++ code being a mandatory argument, the exact same as `build_module`.

16.4.4 `inline_with_numpy`

The difference between this function and the `inline` function is that C-arrays can be used. This means that the necessary arguments (`init_code`, `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

16.4.5 `inline_module_with_numpy`

The difference between this function and the `inline_module` function is that C-arrays can be used. This means that the necessary arguments (`init_code`, `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

16.4.6 import_module

This function can be used to import cached modules from the current work directory or the Instant cache. It has one mandatory argument, `moduleid`, and one keyword argument `cache_dir`. If the latter is given, Instant searches the specified directory instead of the Instant cache, if this directory exists. If the module is not found, `None` is returned. The `moduleid` arguments can be either the module name, a signature, or an object with a function `signature`.

Using the module name or signature, assuming the module `instant_ext` exists in the current working directory or the Instant cache, we import a module in the following way:

```
instant_ext = import_module('instant_ext')
```

Using an object as argument, assuming this object includes a function `signature()` and the module is located in the directory `/tmp`:

```
instant_ext = import_module(signature_object, '/tmp')
```

The imported module, if found, is also placed in the memory cache.

16.4.7 header_and_libs_from_pkgconfig

This function returns a list of include files, flags, libraries and library directories obtain from a pkg-config (pkg-config software package) file. It takes any number of arguments, one string for every package name. It returns four or five arguments. Unless the keyword argument `returnLinkFlags` is given with the value `True`, it returns lists with the include directories, the compile flags, the libraries, and the library directories of the package names given as arguments. If `returnLinkFlags` is `True`, the link flags are returned as a fifth list. Let's look at an example:

```
inc_dirs, comp_flags, libs, lib_dirs, link_flags = \
header_and_libs_from_pkgconfig('ufc-1', 'libxml-2.0',
                               'numpy-1',
                               returnLinkFlags=True)
```

This makes it a easy to write C code that makes use of a package providing a pkg-config file, as we can use the returned lists for compiling and linking our module correctly.

16.4.8 `get_status_output`

This function provides a platform-independent way of running processes in the terminal and extracting the output using the Python module `subprocess`[4]. The one mandatory argument is the command we want to run. Further, there are three keyword arguments. The first is `input`, which should be a string containing input to the process once it is running. The other two are `cwd` and `env`. We refer to the documentation of `subprocess` for a more detailes description of these, but in short the first is the directory in which the process should be executed, while the second is used for setting the necessary environment variables.

16.4.9 `get_swig_version`

Returns the SWIG version installed on the system as a string, for instance '1.3.36'. Accepts no arguments.

16.4.10 `check_swig_version`

Takes a single argument, which should be a string on the same format as the output of `get_swig_version`. Returns `True` if the version of the installed SWIG is equal or greater than the version passed to the function. It also has one keyword argument, `same`. If it is `True`, the function returns `True` if and only if the two versions are the same.

[4]http://docs.python.org/library/subprocess.html

CHAPTER 17

SyFi: Symbolic Construction of Finite Element Basis Functions

By Martin S. Alnæs and Kent-Andre Mardal

Chapter ref: **[alnes-3]**

SyFi is a C++ library for definition of finite elements based on symbolic computations. By solving linear systems of equations symbolically, symbolic expressions for the basis functions of a finite element can be obtained. SyFi contains a collection of such elements.

The SyFi Form Compiler, SFC, is a Python module for generation of fi- nite element code based on symbolic computations. Using equations in UFL format as input and basis functions from SyFi, SFC can generate C++ code which implements the UFC interface for computation of the discretized element tensors. SFC supports generating code based on quadrature or using symbolic integration prior to code generation to produce highly optimized code.

CHAPTER 18

UFC: A Finite Element Code Generation Interface

By Martin S. Alnæs, Anders Logg and Kent-Andre Mardal

Chapter ref: **[alnes-2]**

When combining handwritten libraries with automatically generated code like we do in FEniCS, it is important to have clear boundaries between the two. This is best done by having the generated code implement a fixed interface, such that the library and generated code can be as independent as possible. Such an interface is specified in the project Unified Form-assembly Code (UFC) for finite elements and discrete variational forms. This interface consists of a small set of abstract classes in a single header file, which is well documented. The details of the UFC interface should rarely be visible to the end-user, but can be important for developers and technical users to understand how FEniCS projects fit together. In this chapter we discuss the main design ideas behind the UFC interface, including current limitations and possible future improvements.

CHAPTER 19

UFL: A Finite Element Form Language

By Martin Sandve Alnæs

Chapter ref: **[alnes-1]**

► Editor note: *Sort out what to do with all UFL specific macros and bold math fonts.*

The Unified Form Language – UFL (**??**) – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

The FEniCS project (FEniCS, **?**, **?**) provides a framework for building applications for solving partial differential equations (PDEs). UFL is one of the core components of this framework. It defines the language you *express* your PDEs in. It is the input language and front-end of the form compilers FFC (**??????**) and SFC (**??**). The UFL implementation provides algorithms that the form compilers can use to simplify the compilation process. The output from these form compilers is UFC (**???**) conforming C++ (**?**) code. This code can be used with the C++ library DOLFIN[1] (**???**) to efficiently assemble linear systems and compute solution to PDEs.

The combination of domain specific languages and symbolic computing with finite element methods has been pursued from other angles in several other projects. Sundance (**???**) implements a symbolic engine directly in C++ to define variational forms, and has support for automatic differentiation. The Life (**??**) project uses a domain specific language embedded in C++, based on expression template techniques to specify variational forms. SfePy (**?**) uses SymPy as a

[1]Note that in PyDOLFIN, some parts of UFL is wrapped to blend in with other software components and make the compilation process hidden from the user. This is not discussed here.

symbolic engine, extending it with finite element methods. GetDP (**??**) is another project using a domain specific language for variational forms. The Mathematica package AceGen (**??**) uses the symbolic capabilities of Mathematica to generate efficient code for finite element methods. All these packages have in common a focus on high level descriptions of partial differential equations to achive higher human efficiency in the development of simulation software.

UFL almost resembles a library for symbolic computing, but its scope, goals and priorities are different from generic symbolic computing projects such as GiNaC (**??**), swiginac (Swiginac Python interface to GiNaC) and SymPy (**?**). Intended as a domain specific language and form compiler frontend, UFL is not suitable for large scale symbolic computing.

This chapter is intended both for the FEniCS user who wants to learn how to express her equations, and for other FEniCS developers and technical users who wants to know how UFL works on the inside. Therefore, the sections of this chapter are organized with an increasing amount of technical details. Sections 19.1-19.5 give an overview of the language as seen by the end-user and is intended for all audiences. Sections 19.6-19.9 explain the design of the implementation and dive into some implementation details. Many details of the language has to be omitted in a text such as this, and we refer to the UFL manual (**?**) for a more thorough description. Note that this chapter refers to UFL version 0.3, and both the user interface and the implementation may change in future versions.

Starting with a brief overview, we mention the main design goals for UFL and show an example implementation of a non-trivial PDE in Section 19.1. Next we will look at how to define finite element spaces in Section 19.2, followed by the overall structure of forms and their declaration in Section 19.3. The main part of the language is concerned with defining expressions from a set of data types and operators, which are discussed in Section 19.4. Operators applying to entire forms is the topic of Section 19.5.

The technical part of the chapter begins with Section 19.6 which discusses the representation of expressions. Building on the notation and data structures defined there, how to compute derivatives is discussed in Section 19.7. Some central internal algorithms and key issues in their implementation are discussed in Section 19.8. Implementation details, some of which are specific to the programming language Python (Python programming language), is the topic of Section 19.9. Finally, Section 19.10 discusses future prospects of the UFL project.

19.1 Overview

19.1.1 Design goals

UFL is a unification, refinement and reimplementation of the form languages used in previous versions of FFC and SFC. The development of this language has been motivated by several factors, the most important being:

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL fulfils all these requirements, and by this it represents a major step forward in the capabilities of the FEniCS project.

Tensor algebra and index notation support is modeled after the FFC form language and generalized further. Several nonlinear operators and functions which only SFC supported before have been included in the language. Differentiation of expressions and forms has become an integrated part of the language, and is much easier to use than the way these features were implemented in SFC before. In summary, UFL combines the best of FFC and SFC in one unified form language and adds additional capabilities.

The efficiency of code generated by the new generation of form compilers based on UFL has been verified to match previous form compiler benchmarks (**??**). The form compilation process is now fast enough to blend into the regular application build process. Complicated forms that previously required too much memory to compile, or took tens of minutes or even hours to compile, now compiles in seconds with both SFC and FFC.

19.1.2 Motivational example

One major motivating example during the initial development of UFL has been the equations for elasticity with large deformations. In particular, models of biological tissue use complicated hyperelastic constitutive laws with anisotropies and strong nonlinearities. To implement these equations with FEniCS, all three design goals listed above had to be adressed. Below, one version of the hyperelasticity equations and their corresponding UFL implementation is shown. Keep in mind that this is only intended as an illustration of the close correspondence between the form language and the natural formulation of the equations. The meaning of equations is not necessary for the reader to understand. Note that many other examples are distributed together with UFL.

In the formulation of the hyperelasticity equations presented here, the unknown function is the displacement vector field u. The material coefficients c_1 and c_2 are scalar constants. The second Piola-Kirchoff stress tensor S is computed from the strain energy function $W(\mathbf{C})$. W defines the constitutive law, here a simple Mooney-Rivlin law. The equations relating the displacement and stresses read:

$$
\begin{aligned}
\mathbf{F} &= \mathbf{I} + (\nabla \mathbf{u})^T, \\
\mathbf{C} &= \mathbf{F}^T \mathbf{F}, \\
I_C &= \mathrm{tr}(\mathbf{C}), \\
II_C &= \frac{1}{2}(\mathrm{tr}(\mathbf{C})^2 - \mathrm{tr}(\mathbf{CC})), \\
W &= c_1(I_C - 3) + c_2(II_C - 3), \\
\mathbf{S} &= 2\frac{\partial W}{\partial \mathbf{C}}, \\
\mathbf{P} &= \mathbf{FS}.
\end{aligned}
\tag{19.1}
$$

Approximating the displacement field as $\mathbf{u} = \sum_k u_k \boldsymbol{\phi}_k^1$, the weak forms of the equations are as follows (ignoring boundary conditions):

$$
L(\boldsymbol{\phi}^0; \mathbf{u}, c_1, c_2) = \int_\Omega \mathbf{P} : (\nabla \boldsymbol{\phi}^0)^T \,\mathrm{d}x, \tag{19.2}
$$

$$
a(\boldsymbol{\phi}^0, \boldsymbol{\phi}_k^1; \mathbf{u}, c_1, c_2) = \frac{\partial L}{\partial u_k}. \tag{19.3}
$$

Figure 19.1.2 shows an implementation of these equations in UFL. Notice the close relation between the mathematical notation and the UFL source code. In particular, note the automated differentiation of both the constitutive law and the residual equation. This means a new material law can be implemented by simply changing W, the rest is automatic. In the following sections, the notation, definitions and operators used in this implementation are explained.

19.2 Defining finite element spaces

A polygonal cell is defined by a basic shape and a degree[2], and is declared

```
cell = Cell(shape, degree)
```

UFL defines a set of valid polygonal cell shapes: “interval”, “triangle”, “tetrahedron”, “quadrilateral”, and “hexahedron”. Linear cells of all basic shapes are predefined and can be used instead by writing

```
cell = tetrahedron
```

[2]Note that at the time of writing, the other components of FEniCS does not yet handle higher degree cells.

```
# Finite element spaces
cell = tetrahedron
element = VectorElement("CG", cell, 1)

# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Function(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient Fij = dXi/dxj
I = Identity(cell.d)
F = I + grad(u).T

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0).T)*dx
a = derivative(L, u, phi1)
```

Figure 19.1: UFL implementation of hyperelasticity equations with a Mooney-Rivlin material law.

In the rest of this chapter, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible.

UFL defines syntax for *declaring* finite element spaces, but does not know anything about the actual polynomial basis or degrees of freedom. The polynomial basis is selected implicitly by choosing among predefined basic element families and providing a polynomial degree, but UFL only assumes that there *exists* a basis with a fixed ordering for each finite element space V_h, i.e.

$$V_h = \mathrm{span}\,\{\phi_j\}_{j=1}^{n}\,. \tag{19.4}$$

Basic scalar elements can be combined to form vector elements or tensor elements, and elements can easily be combined in arbitrary mixed element hierarchies.

The set of predefined[3] element family names in UFL includes "Lagrange" (short name "CG"), representing scalar Lagrange finite elements (continuous piecewise polynomial functions), "Discontinuous Lagrange" (short name "DG"), representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions), and a range of other families that can be found in the manual. Each family name has an associated short name for convenience. To print all valid families to screen from Python, call `show_elements()`.

The syntax for declaring elements is best explained with some examples.

```
cell = tetrahedron

P = FiniteElement("Lagrange", cell, 1)
V = VectorElement("Lagrange", cell, 2)
T = TensorElement("DG", cell, 0, symmetry=True)

TH = V + P
ME = MixedElement(T, V, P)
```

In the first line a polygonal cell is selected from the set of predefined linear cells. Then a scalar linear Lagrange element `P` is declared, as well as a quadratic vector Lagrange element `V`. Next a symmetric rank 2 tensor element `T` is defined, which is also piecewise constant on each cell. The code pproceeds to declare a mixed element `TH`, which combines the quadratic vector element `V` and the linear scalar element `P`. This element is known as the Taylor-Hood element. Finally another mixed element with three sub elements is declared. Note that writing `T + V + P` would not result in a mixed element with three direct sub elements, but rather `MixedElement(MixedElement(T + V), P)`.

[3]Form compilers can register additional element families.

19.3 Defining forms

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$,

$$a(v, u; w) = \int_\Omega w \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{19.5}$$

$$L(v; f, g, h) = \int_\Omega f v \,\mathrm{d}x + \int_{\partial\Omega_0} g^2 v \,\mathrm{d}s + \int_{\partial\Omega_1} h v \,\mathrm{d}s. \tag{19.6}$$

These forms can be expressed in UFL as

```
a = dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

where multiplication by the measures `dx`, `ds(0)` and `ds(1)` represent the integrals $\int_{\Omega_0}(\cdot)\,\mathrm{d}x$, $\int_{\partial\Omega_0}(\cdot)\,\mathrm{d}s$, and $\int_{\partial\Omega_1}(\cdot)\,\mathrm{d}s$ respectively.

Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. Considering the above example, the bilinear form a with one coefficient function w is assumed to be evaluated at a later point with a range of basis functions and the coefficient function fixed, that is

$$V_h^1 = \operatorname{span}\left\{\phi_k^1\right\}, \quad V_h^2 = \operatorname{span}\left\{\phi_k^2\right\}, \quad V_h^3 = \operatorname{span}\left\{\phi_k^3\right\}, \tag{19.7}$$

$$w = \sum_{k=1}^{|V_h^2|} w_k \phi_k^3, \quad \{w_k\} \text{ given}, \tag{19.8}$$

$$A_{ij} = a(\phi_i^1, \phi_j^2; w), \quad i = 1, \ldots, |V_h^1|, \quad j = 1, \ldots, |V_h^2|. \tag{19.9}$$

In general, UFL is designed to express forms of the following generalized form:

$$a(\phi^1, \ldots, \phi^r; w^1, \ldots, w^n) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \,\mathrm{d}s + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \,\mathrm{d}S. \tag{19.10}$$

Most of this chapter deals with ways to define the integrand expressions I_k^c, I_k^e and I_k^i. The rest of the notation will be explained below.

The form arguments are divided in two groups, the basis functions $\phi^1, \ldots, \phi^r$ and the coefficient functions $w^1, \ldots, w^n$. All $\{\phi^k\}$ and $\{w^k\}$ are functions in some discrete function space with a basis. Note that the actual basis functions $\{\phi_j^k\}$ and the coefficients $\{w_k\}$ are never known to UFL, but we assume that the ordering of the basis for each finite element space is fixed. A fixed ordering only matters when differentiating forms, explained in Section 19.7.

Each term of a valid form expression must be a scalar-valued expression integrated exactly once, and they must be linear in $\{\phi^k\}$. Any term may have nonlinear dependencies on coefficient functions. A form with one or two basis function arguments ($r = 1, 2$) is called a linear or bilinear form respectively, ignoring its dependency on coefficient functions. These will be assembled to vectors and matrices when used in an application. A form depending only on coefficient functions ($r = 0$) is called a functional, since it will be assembled to a real number.

The entire domain is denoted Ω, the external boundary is denoted $\partial\Omega$, while the set of interior facets of the triangulation is denoted Γ. Sub domains are marked with a suffix, e.g., $\Omega_k \subset \Omega$. As mentioned above, integration is expressed by multiplication with a measure, and UFL defines the measures `dx`, `ds` and `dS`. In summary, there are three kinds of integrals with corresponding UFL representations

- $\int_{\Omega_k} (\cdot) \, \mathrm{d}x \leftrightarrow (\cdot)$`*dx(k)`, called a *cell integral*,
- $\int_{\partial\Omega_k} (\cdot) \, \mathrm{d}s \leftrightarrow (\cdot)$`*ds(k)`, called an *exterior facet integral*,
- $\int_{\Gamma_k} (\cdot) \, \mathrm{d}S \leftrightarrow (\cdot)$`*dS(k)`, called an *interior facet integral*,

Defining a different quadrature order for each term in a form can be achieved by attaching meta data to measure objects, e.g.,

```
dx02 = dx(0, { "integration_order": 2 })
dx14 = dx(1, { "integration_order": 4 })
dx12 = dx(1, { "integration_order": 2 })
L = f*v*dx02 + g*v*dx14 + h*v*dx12
```

Meta data can also be used to override other form compiler specific options separately for each term. For more details on this feature see the manuals of UFL and the form compilers.

19.4 Defining expressions

Most of UFL deals with how to declare expressions such as the integrand expressions in Equation 19.10. The most basic expressions are terminal values, which do not depend on other expressions. Other expressions are called operators, which are discussed in sections 19.4.2-19.4.5.

Terminal value types in UFL include form arguments (which is the topic of Section 19.4.1), geometric quantities, and literal constants. Among the literal constants are scalar integer and floating point values, as well as the d by d identity matrix `I = Identity(d)`. To get unit vectors, simply use rows or columns of the identity matrix, e.g., `e0 = I[0,:]`. Similarly, `I[i,j]` represents the Dirac delta function δ_{ij} (see Section 19.4.2 for details on index notation). Available geometric values are the spatial coordinates $\mathrm{x} \leftrightarrow$ `cell.x` and the facet normal $\mathrm{n} \leftrightarrow$ `cell.n`. The geometric dimension is available as `cell.d`.

19.4.1 Form arguments

Basis functions and coefficient functions are represented by `BasisFunction` and `Function` respectively. The ordering of the arguments to a form is decided by the order in which the form arguments were declared in the UFL code. Each basis function argument represents any function in the basis of its finite element space

$$\phi^j \in \{\phi^j_k\}, \quad V^j_h = \text{span}\left\{\phi^j_k\right\}. \tag{19.11}$$

with the intention that the form is later evaluated for all ϕ_k such as in equation (19.9). Each coefficient function w represents a discrete function in some finite element space V_h; it is usually a sum of basis functions $\phi_k \in V_h$ with coefficients w_k

$$w = \sum_{k=1}^{|V_h|} w_k \phi_k. \tag{19.12}$$

The exception is coefficient functions that can only be evaluated pointwise, which are declared with a finite element with family “Quadrature”. Basis functions are declared for an arbitrary element as in the following manner:

```
phi = BasisFunction(element)
v = TestFunction(element)
u = TrialFunction(element)
```

By using `TestFunction` and `TrialFunction` in declarations instead of `BasisFunction` you can ignore their relative ordering. The only time `BasisFunction` is needed is for forms of arity $r > 2$.

Coefficient functions are declared similarly for an arbitrary element, and shorthand notation exists for declaring piecewise constant functions:

```
w = Function(element)
c = Constant(cell)
v = VectorConstant(cell)
M = TensorConstant(cell)
```

If a form argument u in a mixed finite element space $V_h = V_h^0 \times V_h^1$ is desired, but the form is more easily expressed using sub functions $u_0 \in V_h^0$ and $u_1 \in V_h^1$, you can split the mixed function or basis function into its sub functions in a generic way using `split`:

```
V = V0 + V1
u = Function(V)
u0, u1 = split(u)
```

The `split` function can handle arbitrary mixed elements. Alternatively, a handy shorthand notation for argument declaration followed by `split` is

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Functions(V)
```

19.4.2 Index notation

UFL allows working with tensor expressions of arbitrary rank, using both tensor algebra and index notation. A basic familiarity with tensor algebra and index notation is assumed. The focus here is on how index notation is expressed in UFL.

Assuming a standard orthonormal Euclidean basis $\langle \mathrm{e}_k \rangle_{k=1}^{d}$ for $\mathbb{R}^d$, a vector can be expressed with its scalar components in this basis. Tensors of rank two can be expressed using their scalar components in a dyadic basis $\{\mathrm{e}_i \otimes \mathrm{e}_j\}_{i,j=1}^{d}$. Arbitrary rank tensors can be expressed the same way, as illustrated here.

$$\mathbf{v} = \sum_{k=1}^{d} v_k \mathbf{e}_k, \tag{19.13}$$

$$\mathbf{A} = \sum_{i=1}^{d} \sum_{j=1}^{d} A_{ij} \mathbf{e}_i \otimes \mathbf{e}_j, \tag{19.14}$$

$$\mathcal{C} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k} C_{ijk} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k. \tag{19.15}$$

Here, v, A and $\mathcal{C}$ are rank 1, 2 and 3 tensors respectively. Indices are called *free* if they have no assigned value, such as i in v_i, and *fixed* if they have a fixed value such as 1 in v_1. An expression with free indices represents any expression you can get by assigning fixed values to the indices. The expression A_{ij} is scalar valued, and represents any component (i, j) of the tensor A in the Euclidean basis. When working on paper, it is easy to switch between tensor notation (A) and index notation (A_{ij}) with the knowledge that the tensor and its components are different representations of the same physical quantity. In a programming language, we must express the operations mapping from tensor to scalar components and back explicitly. Mapping from a tensor to its components, for a rank 2 tensor defined as

$$A_{ij} = \mathbf{A} : (\mathbf{e}_i \otimes \mathbf{e}_j), \tag{19.16}$$

$$\tag{19.17}$$

is accomplished using indexing with the notation `A[i,j]`. Defining a tensor $\mathbf{A}$ from component values A_{ij} is defined as

$$\mathbf{A} = A_{ij}\mathbf{e}_i \otimes \mathbf{e}_j, \tag{19.18}$$

and is accomplished using the function `as_vector(Aij, (i,j))`. To illustrate, consider the outer product of two vectors $\mathbf{A} = \mathbf{u} \otimes \mathbf{v} = u_i v_j \mathbf{e}_i \otimes \mathbf{e}_j$, and the corresponding scalar components A_{ij}. One way to implement this is

```
A = outer(u, v)
Aij = A[i, j]
```

Alternatively, the components of $\mathbf{A}$ can be expressed directly using index notation, such as $A_{ij} = u_i v_j$. A_{ij} can then be mapped to $\mathbf{A}$ in the following manner:

```
Aij = v[j]*u[i]
A = as_tensor(Aij, (i, j))
```

These two pairs of lines are mathematically equivalent, and the result of either pair is that the variable `A` represents the tensor $\mathbf{A}$ and the variable `Aij` represents the tensor A_{ij}. Note that free indices have no ordering, so their order of appearance in the expression `v[j]*u[i]` is insignificant. Instead of `as_tensor`, the specialized functions `as_vector` and `as_matrix` can be used. Although a rank two tensor was used for the examples above, the mappings generalize to arbitrary rank tensors.

When indexing expressions, fixed indices can also be used such as in `A[0,1]` which represents a single scalar component. Fixed indices can also be mixed with free indices such as in `A[0,i]`. In addition, slices can be used in place of an index. An example of using slices is `A[0,:]` which is is a vector expression that represents row 0 of `A`. To create new indices, you can either make a single one or make several at once:

```
i = Index()
j, k, l = indices(3)
```

A set of indices `i`, `j`, `k`, `l` and `p`, `q`, `r`, `s` are predefined, and these should suffice for most applications.

If your components are not represented as an expression with free indices, but as separate unrelated scalar expressions, you can build a tensor from them using `as_tensor` and its peers. As an example, lets define a 2D rotation matrix and rotate a vector expression by $\frac{\pi}{2}$:

```
th = pi/2
A = as_matrix([[ cos(th), -sin(th)],
               [ sin(th),  cos(th)]])
u = A*v
```

When indices are repeated in a term, summation over those indices is implied in accordance with the Einstein convention. In particular, indices can be repeated when indexing a tensor of rank two or higher (`A[i,i]`), when differentiating an expression with a free index (`v[i].dx(i)`), or when multiplying two expressions with shared free indices (`u[i]*v[i]`).

$$A_{ii} \equiv \sum_i A_{ii}, \qquad v_i u_i \equiv \sum_i v_i u_i, \qquad v_{i,i} \equiv \sum_i v_{i,i}. \tag{19.19}$$

An expression `Aij = A[i,j]` is represented internally using the `Indexed` class. `Aij` will reference `A`, keeping the representation of the original tensor expression `A` unchanged. Implicit summation is represented explicitly in the expression tree using the class `IndexSum`. Many algorithms become easier to implement with this explicit representation, since e.g. a `Product` instance can never implicitly represent a sum. More details on representation classes are found in Section 19.6.

19.4.3 Algebraic operators and functions

UFL defines a comprehensive set of operators that can be used for composing expressions. The elementary algebraic operators `+`, `-`, `*`, `/` can be used between most UFL expressions with a few limitations. Division requires a scalar expression with no free indices in the denominator. The operands to a sum must have the same shape and set of free indices.

The multiplication operator `*` is valid between two scalars, a scalar and any tensor, a matrix and a vector, and two matrices. Other products could have been defined, but for clarity we use tensor algebra operators and index notation for those rare cases. A product of two expressions with shared free indices implies summation over those indices, see Section 19.4.2 for more about index notation.

Three often used operators are `dot(a, b)`, `inner(a, b)`, and `outer(a, b)`. The dot product of two tensors of arbitrary rank is the sum over the last index of the first tensor and the first index of the second tensor. Some examples are

$$\mathbf{v} \cdot \mathbf{u} = v_i u_i, \tag{19.20}$$

$$\mathbf{A} \cdot \mathbf{u} = A_{ij} u_j \mathbf{e}_i, \tag{19.21}$$

$$\mathbf{A} \cdot \mathbf{B} = A_{ik} B_{kj} \mathbf{e}_i \mathbf{e}_j, \tag{19.22}$$

$$\mathcal{C} \cdot \mathbf{A} = C_{ijk} A_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_l. \tag{19.23}$$

The inner product is the sum over all indices, for example

$$\mathbf{v} : \mathbf{u} = v_i u_i, \tag{19.24}$$

$$\mathbf{A} : \mathbf{B} = A_{ij} B_{ij}, \tag{19.25}$$

$$\mathcal{C} : \mathcal{D} = C_{ijkl} D_{ijkl}. \tag{19.26}$$

Some examples of the outer product are

$$\mathbf{v} \otimes \mathbf{u} = v_i u_j \mathbf{e}_i \mathbf{e}_j, \tag{19.27}$$

$$\mathbf{A} \otimes \mathbf{u} = A_{ij} u_k \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k, \tag{19.28}$$

$$\mathbf{A} \otimes \mathbf{B} = A_{ij} B_{kl} \mathbf{e}_i \mathbf{e}_j \mathbf{e}_k \mathbf{e}_l \tag{19.29}$$

Other common tensor algebra operators are `cross(u,v)`, `transpose(A)` (or `A.T`), `tr(A)`, `det(A)`, `inv(A)`, `cofac(A)`, `dev(A)`, `skew(A)`, and `sym(A)`. Most of these tensor algebra operators expect tensors without free indices. The detailed definitions of these operators are found in the manual.

A set of common elementary functions operating on scalar expressions without free indices are included, in particular `abs(f)`, `pow(f, g)`, `sqrt(f)`, `exp(f)`, `ln(f)`, `sin(f)`, `cos(f)`, and `sign(f)`.

19.4.4 Differential operators

UFL implements derivatives w.r.t. three different kinds of variables. The most used kind is spatial derivatives. Expressions can also be differentiated w.r.t. arbitrary user defined variables. And the final kind of derivatives are derivatives of a form or functional w.r.t. the coefficients of a `Function`. Form derivatives are explained in Section 19.5.1.

Note that derivatives are not computed immediately when declared. A discussion of how derivatives are computed is found in Section 19.7.

Spatial derivatives

Basic spatial derivatives $\frac{\partial f}{\partial x_i}$ can be expressed in two equivalent ways:

```
df = Dx(f, i)
df = f.dx(i)
```

Here, `df` represents the derivative of `f` in the spatial direction x_i. The index `i` can either be an integer, representing differentiation in one fixed spatial direction x_i, or an `Index`, representing differentiation in the direction of a free index. The notation `f.dx(i)` is intended to mirror the index notation $f_{,i}$, which is shorthand for $\frac{\partial f}{\partial x_i}$. Repeated indices imply summation, such that the divergence of a vector can be written $v_{i,i}$, or `v[i].dx(i)`.

Several common compound spatial derivative operators are defined, namely `div`, `grad`, `curl` and `rot` (rot is a synonym for curl). The definition of these operators in UFL follow from the vector of partial derivatives

$$\nabla \equiv \mathbf{e}_k \frac{\partial}{\partial x_k}, \tag{19.30}$$

and the definition of the dot product, outer product, and cross product. Hence,

$$\operatorname{div}(\mathcal{C}) \equiv \nabla \cdot \mathcal{C}, \tag{19.31}$$

$$\operatorname{grad}(\mathcal{C}) \equiv \nabla \otimes \mathcal{C}, \tag{19.32}$$

$$\operatorname{curl}(\mathbf{v}) \equiv \nabla \times \mathbf{v}. \tag{19.33}$$

Note that there are two common ways to define grad and div. This way of defining these operators correspond to writing the convection term from, e.g., the Navier-Stokes equations as

$$\mathbf{w} \cdot \nabla \mathbf{u} = (\mathbf{w} \cdot \nabla)\mathbf{u} = \mathbf{w} \cdot (\nabla \mathbf{u}) = w_i u_{j,i}, \tag{19.34}$$

which is expressed in UFL as

```
dot(w, grad(u))
```

Another illustrative example is the anisotropic diffusion term from, e.g., the bidomain equations, which reads

$$(\mathbf{A}\nabla u) \cdot \mathbf{v} = A_{ij} u_{,j} v_i, \tag{19.35}$$

and is expressed in UFL as

```
dot(A*grad(u), v)
```

In other words, the divergence sums over the *first* index of its operand, and the gradient *prepends* an axis to the tensor shape of its operand. The above definition of curl is only valid for 3D vector expressions. For 2D vector and scalar expressions the definitions are:

$$\operatorname{curl}(\mathbf{u}) \equiv u_{1,0} - u_{0,1}, \tag{19.36}$$

$$\operatorname{curl}(f) \equiv f_{,1}\mathbf{e}_0 - f_{,0}\mathbf{e}_1. \tag{19.37}$$

User defined variables

The second kind of differentiation variables are user-defined variables, which can represent arbitrary expressions. Automating derivatives w.r.t. arbitrary quantities is useful for several tasks, from differentiation of material laws to computing sensitivities. An arbitrary expression g can be assigned to a variable v. An expression f defined as a function of v can be differentiated f w.r.t. v:

$$v = g, \tag{19.38}$$

$$f = f(v), \tag{19.39}$$

$$h(v) = \frac{\partial f(v)}{\partial v}. \tag{19.40}$$

Setting $g = sin(x_0)$ and $f = e^{v^2}$, gives $h = 2ve^{v^2} = 2\sin(x_0)e^{\sin^2(x_0)}$, which can be implemented as follows:

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

Try running this code in a Python session and print the expressions. The result is

```
>>> print v
var0(sin((x)[0]))
>>> print h
d/d[var0(sin((x)[0]))] (exp((var0(sin((x)[0]))) ** 2))
```

Note that the variable has a label 0 ("var0"), and that `h` still represents the abstract derivative. Section 19.7 explains how derivatives are computed.

19.4.5 Other operators

A few operators are provided for the implementation of discontinuous Galerkin methods. The basic concept is restricting an expression to the positive or negative side of an interior facet, which is expressed simply as `v('+')` or `v('-')` respectively. On top of this, the operators `avg` and `jump` are implemented, defined as

$$\mathrm{avg}(v) = \frac{1}{2}(v^+ + v^-), \tag{19.41}$$

$$\mathrm{jump}(v) = v^+ - v^-. \tag{19.42}$$

These operators can only be used when integrating over the interior facets (`*dS`).

The only control flow construct included in UFL is conditional expressions. A conditional expression takes on one of two values depending on the result of a boolean logic expression. The syntax for this is

```
f = conditional(condition, true_value, false_value)
```

which is interpreted as

$$f = \begin{cases} t, & \text{if condition is true}, \\ f, & \text{otherwise}. \end{cases} \tag{19.43}$$

The condition can be one of

- `lt(a, b)` $\leftrightarrow (a < b)$
- `le(a, b)` $\leftrightarrow (a \leq b)$
- `eq(a, b)` $\leftrightarrow (a = b)$
- `gt(a, b)` $\leftrightarrow (a > b)$
- `ge(a, b)` $\leftrightarrow (a \geq b)$
- `ne(a, b)` $\leftrightarrow (a \neq b)$

19.5 Form operators

Once you have defined some forms, there are several ways to compute related forms from them. While operators in the previous section are used to define expressions, the operators discussed in this section are applied to forms, producing new forms. Form operators can both make form definitions more compact and reduce the chances of bugs since changes in the original form will propagate to forms computed from it automatically. These form operators can be combined arbitrarily; given a semi-linear form only a few lines are needed to compute the action of the adjoint of the Jacobi. Since these computations are done prior to processing by the form compilers, there is no overhead at run-time.

19.5.1 Differentiating forms

The form operator `derivative` declares the derivative of a form w.r.t. coefficients of a discrete function (`Function`). This functionality can be used for example to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method. It can also be applied multiple times, which is useful to derive a linear system from a convex functional, in order to find the function that minimizes the functional. For non-trivial equations such expressions can be tedious to calculate by hand. Other areas in which this feature can be useful include optimal control and inverse methods, as well as sensitivity analysis.

In its simplest form, the declaration of the derivative of a form `L` w.r.t. the coefficients of a function `w` reads

```
a = derivative(L, w, u)
```

The form `a` depends on an additional basis function argument `u`, which must be in the same finite element space as the function `w`. If the last argument is omitted, a new basis function argument is created.

Let us step through an example of how to apply `derivative` twice to a functional to derive a linear system. In the following, V_h is a finite element space with some basis, w is a function in V_h, and f is a functional we want to minimize.

Derived from f is a linear form F, and a bilinear form J.

$$V_h = \operatorname{span}\{\phi_k\}, \tag{19.44}$$

$$w(x) = \sum_{k=1}^{|V_h|} w_k \phi_k(x), \tag{19.45}$$

$$f : V_h \rightarrow \mathbb{R}, \tag{19.46}$$

$$F(\phi_i; w) = \frac{\partial}{\partial w_i} f(w), \tag{19.47}$$

$$J(\phi_i, \phi_j; w) = \frac{\partial}{\partial w_j} F(\phi_i; w). \tag{19.48}$$

For a concrete functional $f(w) = \int_\Omega \frac{1}{2} w^2 \, \mathrm{d}x$, we can implement this as

```
v = TestFunction(element)
u = TrialFunction(element)
w = Function(element)
f = 0.5 * w**2 * dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

This code declares two forms `F` and `J`. The linear form `F` represents the standard load vector `w*v*dx` and the bilinear form `J` represents the mass matrix `u*v*dx`.

Derivatives can also be defined w.r.t. coefficients of a function in a mixed finite element space. Consider the Harmonic map equations derived from the functional

$$f(\mathbf{x}, \lambda) = \int_\Omega \nabla \mathbf{x} : \nabla \mathbf{x} + \lambda \mathbf{x} \cdot \mathbf{x} \, \mathrm{d}x, \tag{19.49}$$

where x is a function in a vector finite element space V_h^d and λ is a function in a scalar finite element space V_h. The linear and bilinear forms derived from the functional in Equation 19.49 have basis function arguments in the mixed space $V_h^d + V_h$. The implementation of these forms with automatic linearization reads

```
Vx = VectorElement("CG", triangle, 1)
Vy = FiniteElement("CG", triangle, 1)
u = Function(Vx + Vy)
x, y = split(u)
f = inner(grad(x), grad(x))*dx + y*dot(x,x)*dx
F = derivative(f, u)
J = derivative(F, u)
```

Note that the functional is expressed in terms of the subfunctions x and y, while the argument to derivative must be the single mixed function u. In this example the basis function arguments to derivative are omitted and thus provided automatically in the right function spaces.

Note that in computing derivatives of forms, we have assumed that

$$\frac{\partial}{\partial w_k} \int_\Omega I \,\mathrm{d}x = \int_\Omega \frac{\partial}{\partial w_k} I \,\mathrm{d}x, \tag{19.50}$$

or in particular that the domain Ω is independent of w. Furthermore, note that there is no restriction on the choice of element in this framework, in particular arbitrary mixed elements are supported.

19.5.2 Adjoint

Another form operator is the adjoint a^* of a bilinear form a, defined as $a^*(u, v) = a(v, u)$, which is similar to taking the transpose of the assembled sparse matrix. In UFL this is implemented simply by swapping the test and trial functions, and can be written:

```
a = inner(M*grad(u), grad(v))*dx
ad = adjoint(a)
```

which corresponds to

$$a(M; v, u) = \int_\Omega (\mathbf{M}\nabla\mathbf{u}) : \nabla\mathbf{v} \,\mathrm{d}x = \int_\Omega M_{ik} u_{j,k} v_{j,i} \,\mathrm{d}x, \tag{19.51}$$

$$a^*(M; v, u) = a(M; u, v) = \int_\Omega (\mathbf{M}\nabla\mathbf{v}) : \nabla\mathbf{u} \,\mathrm{d}x. \tag{19.52}$$

This automatic transformation is particularly useful if we need the adjoint of nonsymmetric bilinear forms computed using derivative, since the explicit expressions for a are not at hand. Several of the form operators below are most useful when used in conjunction with derivative.

19.5.3 Replacing functions

Evaluating a form with new definitions of form arguments can be done by replacing terminal objects with other values. Lets say you have defined a form L that depends on some functions f and g. You can then specialize the form by replacing these functions with other functions or fixed values, such as

$$L(f, g; v) = \int_\Omega (f^2/(2g)) v \,\mathrm{d}x, \tag{19.53}$$

$$L_2(f, g; v) = L(g, 3; v) = \int_\Omega (g^2/6) v \,\mathrm{d}x. \tag{19.54}$$

This feature is implemented with replace, as illustrated in this case:

```
L = f**2 / (2*g) * v * dx
L2 = replace(L, { f: g, g: 3})
L3 = g**2 / 6 * v * dx
```

Here `L2` and `L3` represents exactly the same form. Since they depend only on `g`, the code generated for these forms can be more efficient.

19.5.4 Action

Sparse matrix-vector multiplication is an important operation in PDE solver applications. In some cases the matrix is not needed explicitly, only the action of the matrix on a vector, the result of the matrix-vector multiplication. You can assemble the action of the matrix on a vector directly by defining a linear form for the action of a bilinear form on a function, simply writing `L = action(a, w)` or `L = a*w`, with `a` any bilinear form and `w` being any `Function` defined on the same finite element as the trial function in `a`.

19.5.5 Splitting a system

If you prefer to write your PDEs with all terms on one side such as

$$a(v,u) - L(v) = 0, \tag{19.55}$$

you can declare forms with both linear and bilinear terms and split the equations afterwards:

```
pde = u*v*dx - f*v*dx
a, L = system(pde)
```

Here `system` is used to split the PDE into its bilinear and linear parts. Alternatively, `lhs` and `rhs` can be used to obtain the two parts separately.

19.5.6 Computing the sensitivity of a function

If you have found the solution u to Equation (19.55), and u depends on some constant scalar value c, you can compute the sensitivity of u w.r.t. changes in c. If u is represented by a coefficient vector x that is the solution to the algebraic linear system $Ax = b$, the coefficients of $\frac{\partial u}{\partial c}$ are $\frac{\partial x}{\partial c}$. Applying $\frac{\partial}{\partial c}$ to $Ax = b$ and using the chain rule, we can write

$$A\frac{\partial x}{\partial c} = \frac{\partial b}{\partial c} - \frac{\partial A}{\partial c}x, \tag{19.56}$$

and thus $\frac{\partial x}{\partial c}$ can be found by solving the same algebraic linear system used to compute x, only with a different right hand side. The linear form corresponding to the right hand side of Equation (19.56) can be written

```
u = Function(element)
sL = diff(L, c) - action(diff(a, c), u)
```

or you can use the equivalent form transformation

```
sL = sensitivity_rhs(a, u, L, c)
```

Note that the solution `u` must be represented by a `Function`, while u in $a(v, u)$ is represented by a `BasisFunction`.

19.6 Expression representation

19.6.1 The structure of an expression

Most of the UFL implementation is concerned with expressing, representing, and manipulating expressions. To explain and reason about expression representations and algorithms operating on them, we need an abstract notation for the structure of an expression. UFL expressions are representations of programs, and the notation should allow us to see this connection without the burden of implementation details.

The most basic UFL expressions are expressions with no dependencies on other expressions, called *terminals*. Other expressions are the result of applying some *operator* to one or more existing expressions. All expressions are immutable; once constructed an expression will never change. Manipulating an expression always results in a new expression being created.

Consider an arbitrary (non-terminal) expression z. This expression depends on a set of terminal values $\{t_i\}$, and is computed using a set of operators $\{f_i\}$. If each subexpression of z is labeled with an integer, an abstract program can be written to compute z by computing a sequence of subexpressions $\langle y_i \rangle_{i=1}^n$ and setting $z = y_n$. Algorithm 6 shows such a program.

Algorithm 6 Program to compute an expression z

```
for $i = 1, \ldots, m$:
    $y_i = t_i =$ terminal expression
for $i = m + 1, \ldots, n$:
    $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
$z = y_n$
```

Each terminal expression $y_i = t_i$ is a literal constant or input arguments to the program. A non-terminal subexpression y_i is the result of applying an operator f_i to a sequence of previously computed expressions $\langle y_j \rangle_{j \in \mathfrak{I}_i}$, where $\mathfrak{I}_i$ is a set of expression labels. Note that the order in which subexpressions are computed can be arbitrarily chosen, except that we require $j < i \forall j \in \mathfrak{I}_i$, such that all dependencies of a subexpression y_i has been computed before y_i. In particular, all terminals are numbered first in this algorithm for notational convenience only.

The program can be represented as a graph, where each expression y_i corresponds to a graph vertex and each direct dependency between two expressions is a graph edge. More formally,

$$G = (V, E), \tag{19.57}$$
$$V = \langle v_i \rangle_{i=1}^{n} = \langle y_i \rangle_{i=1}^{n}, \tag{19.58}$$
$$E = \{e_i\} = \bigcup_{i=1}^{n} \{(i,j) \forall j \in \mathfrak{I}_i\}. \tag{19.59}$$

This graph is clearly directed, since dependencies have a direction. It is acyclic, since an expression can only be constructed from existing expressions and never be modified. Thus we can say that an UFL expression represents a program, and can be represented using a directed acyclic graph (DAG). There are two ways this DAG can be represented in UFL, a linked representation called the expression tree, and a linearized representation called the computational graph.

19.6.2 Tree representation

► Editor note: *Redraw these figures in Inkscape.*

An expression is usually represented as an expression tree. Each subexpression is represented by a tree node, which is the root of a tree of its own. The leaves of the tree are terminal expressions, and operators have their operands as children. An expression tree for the stiffness term $\nabla u : \nabla v$ is illustrated in Figure 19.3. The terminals u and v have no children, and the term ∇u is itself represented by a tree with two nodes. The names in this figure, `Grad`, `Inner` and `BasisFunction`, reflect the names of the classes used in UFL to represent the expression nodes. Taking the gradient of an expression with `grad(u)` gives an expression representation `Grad(u)`, and `inner(a, b)` gives an expression representation `Inner(a, b)`. In general, each expression node is an instance of some subclass of `Expr`. The class `Expr` is the superclass of a hierarchy containing all terminal types and operator types UFL supports. `Expr` has two direct subclasses, `Terminal` and `Operator`, as illustrated in Figure 19.2.

Each expression node represents a single vertex v_i in the DAG. Recall from Algorithm 6 that non-terminals are expressions $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$. The operator

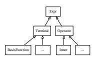


Figure 19.2: Expression class hierarchy.

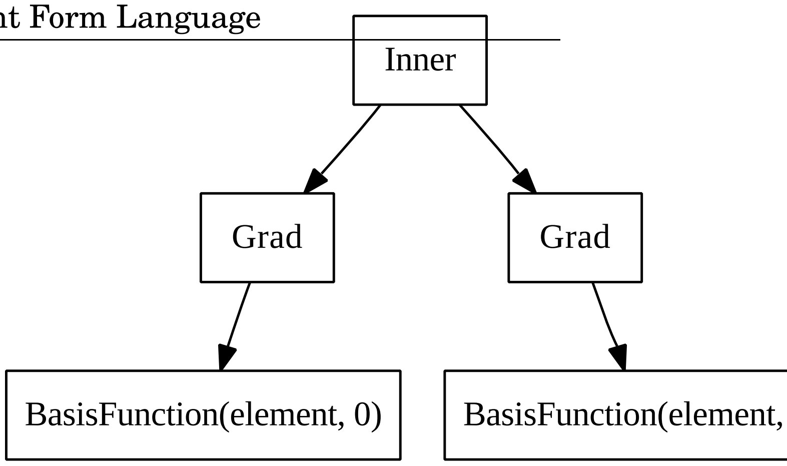


Figure 19.3: Expression tree for $\nabla \mathbf{u} : \nabla \mathbf{v}$.

f_i is represented by the class of the expression node, while the expression y_i is represented by the instance of this class. The edges of the DAG is not stored explicitly in the tree representation. However, from an expression node representing the vertex v_i, a tuple with the vertices $\langle y_j \rangle_{j \in \mathfrak{I}_i}$ can be obtained by calling `yi.operands()`. These expression nodes represent the graph vertices that have edges pointing to them from y_i. Note that this generalizes to terminals where there are no outgoing edges and `t.operands()` returns an empty tuple.

19.6.3 Expression node properties

Any expression node `e` (an `Expr` instance) has certain generic properties, and the most important ones will be explained here. Above it was mentioned that `e.operands()` returns a tuple with the child nodes. Any expression node can be reconstructed with modified operands using `e.reconstruct(operands)`, where `operands` is a tuple of expression nodes. The invariant `e.reconstruct(e.operands()) == e` should always hold. This function is required because expression nodes are immutable, they should never be modified. The immutable property ensures that expression nodes can be reused and shared between expressions without side effects in other parts of a program.

► Editor note*: Stick ugly text sticking out in margin.*

In Section 19.4.2 the tensor algebra and index notation capabilities of UFL was discussed. Expressions can be scalar or tensor-valued, with arbitrary rank and shape. Therefore, each expression node has a value shape `e.shape()`, which is a tuple of integers with the dimensions in each tensor axis. Scalar expressions have shape `()`. Another important property is the set of free indices in an expression, obtained as a tuple using `e.free_indices()`. Although the free indices have no ordering, they are represented with a tuple of `Index` instances for simplicity. Thus the ordering within the tuple carries no meaning.

UFL expressions are referentially transparent with some exceptions. Referential transparency means that a subexpression can be replaced by another representation of its value without changing the meaning of the expression. A key point here is that the value of an expression in this context includes the tensor shape and set of free indices. Another important point is that the derivative of a function $f(v)$ in a point, $f'(v)|_{v=g}$, depends on function values in the vicinity of $v = g$. The effect of this dependency is that operator types matter when differentiating, not only the current value of the differentiation variable. In particular, a `Variable` cannot be replaced by the expression it represents, because `diff` depends on the `Variable` instance and not the expression it has the value of. Similarly, replacing a `Function` with some value will change the meaning of an expression that contains derivatives w.r.t. function coefficients.

The following example illustrate this issue.

```
e = 0
v = variable(e)
f = sin(v)
g = diff(f, v)
```

Here `v` is a variable that takes on the value 0, but `sin(v)` cannot be simplified to 0 since the derivative of `f` then would be 0. The correct result here is `g = cos(v)`.

19.6.4 Linearized graph representation

A linearized representation of the DAG is useful for several internal algorithms, either to achieve a more convenient formulation of an algorithm or for improved performance. UFL includes tools to build a linearized representation of the DAG, the *computational graph*, from any expression tree. The computational graph $G = V, E$ is a data structure based on flat arrays, directly mirroring the definition of the graph in equations (19.57)-(19.59). This simple data structure makes some algorithms easier to implement or more efficient than the recursive tree representation. One array (Python list) `V` is used to store the vertices $\langle v_i \rangle_{i=1}^n$ of the DAG. For each vertex v_i an expression node y_i is stored to represent it. Thus the expression tree for each vertex is also directly available, since each expression node is the root of its own expression tree. The edges are stored in an array `E` with integer tuples `(i,j)` representing an edge from v_i to v_j, i.e. that v_j is an operand of v_i. The graph is built using a post-order traversal, which guarantees that the vertices are ordered such that $j < i \forall j \in \mathfrak{I}_i$.

From the edges E, related arrays can be computed efficiently; in particular the vertex indices of dependencies of a vertex v_i in both directions are useful:

$$\begin{aligned} V_{out} &= \langle \mathfrak{I}_i \rangle_{i=1}^n , \\ V_{in} &= \langle \{ j | i \in \mathfrak{I}_j \} \rangle_{i=1}^n \end{aligned} \tag{19.60}$$

These data structures can be easily constructed for any expression:

```
G = Graph(expression)
V, E = G
Vin = G.Vin()
Vout = G.Vout()
```

A nice property of the computational graph built by UFL is that no two vertices will represent the same identical expression. During graph building, subexpressions are inserted in a hash map (Python dict) to achieve this.

Free indices in expression nodes can complicate the interpretation of the linearized graph when implementing some algorithms. One solution to that can be to apply `expand_indices` before constructing the graph. Note however that free indices cannot be regained after expansion.

19.6.5 Partitioning

UFL is intended as a front-end for form compilers. Since the end goal is generation of code from expressions, some utilities are provided for the code generation process. In principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each operation separately, basically mirroring Algorithm 6. However, a good form compiler should be able to produce better code. UFL provides utilities for partitioning the computational graph into subgraphs (partitions) based on dependencies of subexpressions, which enables quadrature based form compilers to easily place subexpressions inside the right sets of loops. The function `partition` implements this feature. Each partition is represented by a simple array of vertex indices.

19.7 Computing derivatives

When a derivative expression is declared by the end-user of the form language, an expression node is constructed to represent it, but nothing is computed. The type of this expression node is a subclass of `Derivative`. Differential operators cannot be expressed natively in a language such as C++. Before code can be generated from the derivative expression, some kind of algorithm to evaluate derivatives must be applied. Computing exact derivatives is important, which rules out approximations by divided differences. Several alternative algorithms exist for computing exact derivatives. All relevant algorithms are based on the chain rule combined with differentiation rules for each expression node type. The main differences between the algorithms are in the extent of which subexpressions are reused, and in the way subexpressions are accumulated.

Below, the differences and similarities between some of the simplest algorithms are discussed. After the algorithm currently implemented in UFL has been explained, extensions to tensor and index notation and higher order derivatives are discussed. Finally, the section is closed with some remarks about the differentiation rules for terminal expressions.

19.7.1 Relations to form compiler approaches

Before discussing the choice of algorithm for computing derivatives, let us concider the context in which the results will be used. Although UFL does not generate code, some form compiler issues are relevant to this context.

Mixing derivative computation into the code generation strategy of each form compiler would lead to a significant duplication of implementation effort. To separate concerns and keep the code manageable, differentiation is implemented as part of UFL in such a way that the form compilers are independent of the chosen differentiation strategy. Before expressions are interpreted by a form compiler, differential operators should be evaluated such that the only operators left are non-differential operators[4]. Therefore, it is advantageous to use the same representation for the evaluated derivative expressions and other expressions.

The properties of each differentiation algorithm is strongly related to the structure of the expression representation. However, UFL has no control over the final expression representation used by the form compilers. The main difference between the current form compilers is the way in which expressions are integrated. For large classes of equations, symbolic integration or a specialized tensor representation have proven highly efficient ways to evaluate element tensors (**???**). However, when applied to more complex equations, the run-time performance of both these approaches is beaten by code generated with quadrature loops (**??**). To apply symbolic differentiation, polynomials are expanded which destroys the structure of the expressions, gives potential exponential growth of expression sizes, and hides opportunities for subexpression reuse. Similarly, the tensor representation demands a canonical representation of the integral expressions.

In summary, both current non-quadrature form compiler approaches change the structure of the expressions they get from UFL. This change makes the interaction between the differentiation algorithm and the form compiler approach hard to control. However, this will only become a problem for complex equations, in which case quadrature loop based code is more suitable. Code generation using quadrature loops can more easily mirror the inherent structure of UFL expressions.

[4]An exception is made for spatial derivatives of terminals which are unknown to UFL because they are provided by the form compilers.

19.7.2 Approaches to computing derivatives

Algorithms for computing derivatives are designed with different end goals in mind. Symbolic Differentiation (SD) takes as input a single symbolic expression and produces a new symbolic expression for the derivative of the input. Automatic Differentiation (AD) takes as input a program to compute a function and produces a new program to compute the derivative of the function. Several variants of AD algorithms exist, the two most common being Forward Mode AD and Reverse Mode AD (**?**). More advanced algorithms exist, and is an active research topic.is a symbolic expression, represented by an expression tree. But the expression tree is a directed acyclic graph that represents a program to evaluate said expression. Thus it seems the line between SD and AD becomes less distinct in this context.

Naively applied, SD can result in huge expressions, which can both require a lot of memory during the computation and be highly inefficient if written to code directly. However, some illustrations of the inefficiency of symbolic differentiation, such as in (**?**), are based on computing closed form expressions of derivatives in some stand-alone computer algebra system (CAS). Copying the resulting large expressions directly into a computer code can lead to very inefficient code. The compiler may not be able to detect common subexpressions, in particular if simplification and rewriting rules in the CAS has changed the structure of subexpressions with a potential for reuse.

In general, AD is capable of handling algorithms that SD can not. A tool for applying AD to a generic source code must handle many complications such as subroutines, global variables, arbitrary loops and branches (**???**). Since the support for program flow constructs in UFL is very limited, the AD implementation in UFL will not run into such complications. In Section 19.7.3 the similarity between SD and forward mode AD in the context of UFL is explained in more detail.

19.7.3 Forward mode Automatic Differentiation

Recall Algorithm 6, which represents a program for computing an expression z from a set of terminal values $\{t_i\}$ and a set of elementary operations $\{f_i\}$. Assume for a moment that there are no differential operators among $\{f_i\}$. The algorithm can then be extended to compute the derivative $\frac{dz}{dv}$, where v represents a differentiation variable of any kind. This extension gives Algorithm 7.

This way of extending a program to simultaneously compute the expression z and its derivative $\frac{dz}{dv}$ is called forward mode automatic differentiation (AD). By renaming y_i and $\frac{dy_i}{dv}$ to a new sequence of values $\langle \hat{y}_j \rangle_{j=1}^{\hat{n}}$, Algorithm 7 can be rewritten as shown in Algorithm 8, which is isomorphic to Algorithm 6 (they have exactly the same structure).

Since the program in Algorithm 6 can be represented as a DAG, and Algo-

Algorithm 7 Forward mode AD on Algorithm 6

```
for $i = 1, \ldots, m$:
    $y_i = t_i$
    $\frac{dy_i}{dv} = \frac{dt_i}{dv}$
for $i = m + 1, \ldots, n$:
    $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
    $\frac{dy_i}{dv} = \sum_{k \in \mathfrak{I}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$
$z = y_n$
$\frac{dz}{dv} = \frac{dy_n}{dv}$
```

Algorithm 8 Program to compute $\frac{dz}{dv}$ produced by forward mode AD

```
for $i = 1, \ldots, \hat{m}$:
    $\hat{y}_i = \hat{t}_i$
for $i = \hat{m} + 1, \ldots, \hat{n}$:
    $\hat{y}_i = \hat{f}_i(\langle \hat{y}_j \rangle_{j \in \hat{\mathfrak{I}}_i})$
$\frac{dz}{dv} = \hat{y}_{\hat{n}}$
```

rithm 8 is isomorphic to Algorithm 6, the program in Algorithm 8 can also be represented as a DAG. Thus a program to compute $\frac{dz}{dv}$ can be represented by an expression tree built from terminal values and non-differential operators.

The currently implemented algorithm for computing derivatives in UFL follows forward mode AD closely. Since the result is a new expression tree, the algorithm can also be called symbolic differentiation. In this context, the differences between the two are implementation details. To ensure that we can reuse expressions properly, simplification rules in UFL avoids modifying the operands of an operator. Naturally repeated patterns in the expression can therefore be detected easily by the form compilers. Efficient common subexpression elimination can then be implemented by placing subexpressions in a hash map. However, there are simplifications such as $0 * f \to 0$ and $1 * f \to f$ which simplify the result of the differentiation algorithm automatically as it is being constructed. These simplifications are crucial for the memory use during derivative computations, and the performance of the resulting program.

19.7.4 Extensions to tensors and indexed expressions

So far we have not considered derivatives of non-scalar expression and expressions with free indices. This issue does not affect the overall algorithms, but it does affect the local derivative rules for each expression type.

Consider the expression `diff(A, B)` with `A` and `B` matrix expressions. The meaning of derivatives of tensors w.r.t. to tensors is easily defined via index notation, which is heavily used within the differentiation rules:

$$\frac{d\mathbf{A}}{d\mathbf{B}} = \frac{dA_{ij}}{dB_{kl}} \mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l \tag{19.61}$$

Derivatives of subexpressions are frequently evaluated to literal constants. For indexed expressions, it is important that free indices are propagated correctly with the derivatives. Therefore, differentiated expressions will some times include literal constants annotated with free indices.

There is one rare and tricky corner case when an index sum binds an index i such as in $(v_i v_i)$ and the derivative w.r.t. x_i is attempted. The simplest example of this is the expression $(v_i v_i)_{,j}$, which has one free index j. If j is replaced by i, the expression can still be well defined, but you would never write $(v_i v_i)_{,i}$ manually. If the expression in the parenthesis is defined in a variable `e = v[i]*v[i]`, the expression `e.dx(i)` looks innocent. However, this will cause problems as derivatives (including the index i) are propagated up to terminals. If this case is encountered it will be detected and an error message will be triggered. To avoid it, simply use different index instances. In the future, this case may be handled by relabeling indices to change this expression into $(v_j v_j)_{,i} u_i$.

19.7.5 Higher order derivatives

A simple forward mode AD implementation such as Algorithm 7 only considers one differentiation variable. Higher order or nested differential operators must also be supported, with any combination of differentiation variables. A simple example illustrating such an expression can be

$$a = \frac{d}{dx}\left(\frac{d}{dx} f(x) + 2\frac{d}{dy} g(x, y)\right). \tag{19.62}$$

Considerations for implementations of nested derivatives in a functional[5] framework have been explored in several papers (**???**).

In the current UFL implementation this is solved in a different fashion. Considering Equation (19.62), the approach is simply to compute the innermost derivatives $\frac{d}{dx} f(x)$ and $\frac{d}{dy} g(x, y)$ first, and then computing the outer derivatives. This approach is possible because the result of a derivative computation is represented as an expression tree just as any other expression. Mainly this approach was chosen because it is simple to implement and easy to verify. Whether other approaches are faster has not been investigated. Furthermore, alternative AD algorithms such as reverse mode can be experimented with in the future without concern for nested derivatives in the first implementations.

An outer controller function `apply_ad` handles the application of a single variable AD routine to an expression with possibly nested derivatives. The AD

[5]Functional as in functional languages.

routine is a function accepting a derivative expression node and returning an expression where the single variable derivative has been computed. This routine can be an implementation of Algorithm 8. The result of `apply_ad` is mathematically equivalent to the input, but with no derivative expression nodes left[6].

The function `apply_ad` works by traversing the tree recursively in post-order, discovering subtrees where the root represents a derivative, and applying the provided AD routine to the derivative subtree. Since the children of the derivative node has already been visited by `apply_ad`, they are guaranteed to be free of derivative expression nodes and the AD routine only needs to handle the case discussed above with algorithms 7 and 8.

The complexity of the `ad_routine` should be $O(n)$, with n being the size of the expression tree. The size of the derivative expression is proportional to the original expression. If there are d derivative expression nodes in the expression tree, the complexity of this algorithm is $O(dn)$, since `ad_routine` is applied to subexpressions d times. As a result the worst case complexity of `apply_ad` is $O(n^2)$, but in practice $d \ll n$. A recursive implementation of this algorithm is shown in Figure 19.4.

```
def apply_ad(e, ad_routine):
    if isinstance(e, Terminal):
        return e
    ops = [apply_ad(o, ad_routine) for o in e.operands()]
    e = e.reconstruct(*ops)
    if isinstance(e, Derivative):
        e = ad_routine(e)
    return e
```

Figure 19.4: Simple implementation of recursive `apply_ad` procedure.

19.7.6 Basic differentiation rules

To implement the algorithm descriptions above, we must implement differentiation rules for all expression node types. Derivatives of operators can be implemented as generic rules independent of the differentiation variable, and these are well known and not mentioned here. Derivatives of terminals depend on the differentiation variable type. Derivatives of literal constants are of course always zero, and only spatial derivatives of geometric quantities are non-zero. Since form arguments are unknown to UFL (they are provided externally by the form compilers), their spatial derivatives ($\frac{\partial \phi^k}{\partial x_i}$ and $\frac{\partial w^k}{\partial x_i}$) are considered input arguments as well. In all derivative computations, the assumption is made that

[6]Except direct spatial derivatives of form arguments, but that is an implementation detail.

form coefficients have no dependencies on the differentiation variable. Two more cases needs explaining, the user defined variables and derivatives w.r.t. the coefficients of a `Function`.

If v is a `Variable`, then we define $\frac{dt}{dv} \equiv 0$ for any terminal t. If v is scalar valued then $\frac{dv}{dv} \equiv 1$. Furthermore, if **V** is a tensor valued `Variable`, its derivative w.r.t. itself is

$$\frac{d\mathbf{V}}{d\mathbf{V}} = \frac{dV_{ij}}{dV_{kl}}\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l = \delta_{ik}\delta_{jl}\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \mathbf{e}_l. \tag{19.63}$$

In addition, the derivative of a variable w.r.t. something else than itself equals the derivative of the expression it represents:

$$v = g, \tag{19.64}$$
$$\frac{dv}{dz} = \frac{dg}{dz}. \tag{19.65}$$

Finally, we consider the operator `derivative`, which represents differentiation w.r.t. all coefficients $\{w_k\}$ of a function w. Consider an object `element` which represents a finite element space V_h with a basis $\{\phi_k\}$. Next consider form arguments defined in this space:

```
v = BasisFunction(element)
w = Function(element)
```

The BasisFunction instance `v` represents any $v \in \{\phi_k\}$, while the `Function` instance `w` represents the sum

$$w = \sum_k w_k \phi_k(x). \tag{19.66}$$

The derivative of `w` w.r.t. any w_k is the corresponding basis function in V_h,

$$\frac{\partial w}{\partial w_k} = \phi_k, \qquad k = 1, \ldots, |V_h|, \tag{19.67}$$
$$\tag{19.68}$$

which can be represented by `v`, since

$$v \in \langle \phi_k \rangle_{k=1}^{|V_h|} = \left\langle \frac{\partial w}{\partial w_k} \right\rangle_{k=1}^{|V_h|}. \tag{19.69}$$

Note that `v` should be a basis function instance that has not already been used in the form.

19.8 Algorithms

In this section, some central algorithms and key implementation issues are discussed, much of which relates to the Python programming language. Thus, this section is mainly intended for developers and others who need to relate to UFL on a technical level.

19.8.1 Effective tree traversal in Python

Applying some action to all nodes in a tree is naturally expressed using recursion:

```
def walk(expression, pre_action, post_action):
    pre_action(expression)
    for o in expression.operands():
        walk(o)
    post_action(expression)
```

This implementation simultaneously covers pre-order traversal, where each node is visited before its children, and post-order traversal, where each node is visited after its children.

A more "pythonic" way to implement iteration over a collection of nodes is using generators. A minimal implementation of this could be

```
def post_traversal(root):
    for o in root.operands():
        yield post_traversal(o)
    yield root
```

which then enables the natural Python syntax for iteration over expression nodes:

```
for e in post_traversal(expression):
    post_action(e)
```

For efficiency, the actual implementation of `post_traversal` in UFL is not using recursion. Function calls are very expensive in Python, which makes the non-recursive implementation an order of magnitude faster than the above.

19.8.2 Type based function dispatch in Python

▶ Editor note: *Make code fit in box.*

A common task in both symbolic computing and compiler implementation is the selection of some operation based on the type of an expression node. For a

```
class ExampleFunction(MultiFunction):
    def __init__(self):
        MultiFunction.__init__(self)

    def terminal(self, expression):
        return "Got a Terminal subtype %s." % type(expression)

    def operator(self, expression):
        return "Got an Operator subtype %s." % type(expression)

    def basis_function(self, expression):
        return "Got a BasisFunction."

    def sum(self, expression):
        return "Got a Sum."

m = ExampleFunction()

cell = triangle
element = FiniteElement("CG", cell, 1)
x = cell.x
print m(BasisFunction(element))
print m(x)
print m(x[0] + x[1])
print m(x[0] * x[1])
```

Figure 19.5: Example declaration and use of a multifunction

selected few operations, this is done using overloading of functions in the subclasses of `Expr`, but this is not suitable for all operations.

In many cases type-specific operations must be implemented together in the algorithm instead of distributed across class definitions. One way to implement type based operation selection is to use a type switch, or a sequence of if-tests such as this:

```
if isinstance(expression, IntValue):
    result = int_operation(expression)
elif isinstance(expression, Sum):
    result = sum_operation(expression)
# etc.
```

There are several problems with this approach, one of which is efficiency when there are many types to check. A type based function dispatch mechanism with efficiency independent of the number of types is implemented as an alternative through the class `MultiFunction`. The underlying mechanism is a dict lookup (which is $O(1)$) based on the type of the input argument, followed by a call to the function found in the dict. The lookup table is built in the `MultiFunction` constructor. Functions to insert in the table are discovered automatically using the introspection capabilites of Python.

A multifunction is declared as a subclass of `MultiFunction`. For each type that should be handled particularly, a member function is declared in the subclass. The `Expr` classes use the `CamelCaps` naming convention, which is automatically converted to `underscore_notation` for corresponding function names, such as `BasisFunction` and `basis_function`. If a handler function is not declared for a type, the closest superclass handler function is used instead. Note that the `MultiFunction` implementation is specialized to types in the `Expr` class hierarchy. The declaration and use of a multifunction is illustrated in Figure 19.5. Note that `basis_function` and `sum` will handle instances of the exact types `BasisFunction` and `Sum`, while `terminal` and `operator` will handle the types `SpatialCoordinate` and `Product` since they have no specific handlers.

19.8.3 Implementing expression transformations

Many transformations of expressions can be implemented recursively with some type-specific operation applied to each expression node. Examples of operations are converting an expression node to a string representation, an expression representation using an symbolic external library, or an UFL representation with some different properties. A simple variant of this pattern can be implemented using a multifunction to represent the type-specific operation:

```
def apply(e, multifunction):
    ops = [apply(o, multifunction) for o in e.operands()]
    return multifunction(e, *ops)
```

The basic idea is as follows. Given an expression node `e`, begin with applying the transformation to each child node. Then return the result of some operation specialized according to the type of `e`, using the already transformed children as input.

The `Transformer` class implements this pattern. Defining a new algorithm using this pattern involves declaring a `Transformer` subclass, and implementing the type specific operations as member functions of this class just as with `MultiFunction`. The difference is that member functions take one additional argument for each operand of the expression node. The transformed child nodes are supplied as these additional arguments. The following code replaces terminal objects with objects found in a dict `mapping`, and reconstructs operators with the transformed expression trees. The algorithm is applied to an expression by calling the function `visit`, named after the similar Visitor pattern.

```
class Replacer(Transformer):
    def __init__(self, mapping):
        Transformer.__init__(self)
        self.mapping = mapping

    def operator(self, e, *ops):
        return e.reconstruct(*ops)

    def terminal(self, e):
        return self.mapping.get(e, e)

f = Constant(triangle)
r = Replacer({f: f**2})
g = r.visit(2*f)
```

After running this code the result is $g = 2f^2$. The actual implementation of the `replace` function is similar to this code.

In some cases, child nodes should not be visited before their parent node. This distinction is easily expressed using `Transformer`, simply by omitting the member function arguments for the transformed operands. See the source code for many examples of algorithms using this pattern.

19.8.4 Important transformations

There are many ways in which expression representations can be manipulated. Here, we describe a few particularly important transformations. Note that each of these algorithms removes some abstractions, and hence may remove some opportunities for analysis or optimization.

Some operators in UFL are termed “compound” operators, meaning they can be represented by other elementary operators. Try defining an expression `e = inner(grad(u), grad(v))`, and print `repr(e)`. As you will see, the representation of `e` is `Inner(Grad(u), Grad(v))` (with some more details for `u` and `v`). This way the input expressions are easier to recognize in the representation, and rendering of expressions to for example LaTeX format can show the original compound operators as written by the end-user.

However, since many algorithms must implement actions for each operator type, the function `expand_compounds` is used to replace all expression nodes of “compound” types with equivalent expressions using basic types. When this operation is applied to the input forms from the user, algorithms in both UFL and the form compilers can still be written purely in terms of basic operators.

Another important transformation is `expand_derivatives`, which applies automatic differentiation to expressions, recursively and for all kinds of derivatives. The end result is that most derivatives are evaluated, and the only derivative operator types left in the expression tree applies to terminals. The precondition for this algorithm is that `expand_compounds` has been applied.

Index notation and the `IndexSum` expression node type complicate interpretation of an expression tree in some contexts, since free indices in its summand expression will take on multiple values. In some cases, the transformation `expand_indices` comes in handy, the end result of which is that there are no free indices left in the expression. The precondition for this algorithm is that `expand_compounds` and `expand_derivatives` have been applied.

19.8.5 Evaluating expressions

Even though UFL expressions are intended to be compiled by form compilers, it can be useful to evaluate them to floating point values directly. In particular, this makes testing and debugging of UFL much easier, and is used extensively in the unit tests. To evaluate an UFL expression, values of form arguments and geometric quantities must be specified. Expressions depending only on spatial coordinates can be evaluated by passing a tuple with the coordinates to the call operator. The following code from an interactive Python session shows the syntax:

```
>>> cell = triangle
>>> x = cell.x
>>> e = x[0]+x[1]
>>> print e((0.5,0.7))
1.2
```

Other terminals can be specified using a dictionary that maps from terminal instances to values. This code extends the above code with a mapping:

```
c = Constant(cell)
e = c*(x[0]+x[1])
print e((0.5,0.7), { c: 10 })
```

If functions and basis functions depend on the spatial coordinates, the mapping can specify a Python callable instead of a literal constant. The callable must take the spatial coordinates as input and return a floating point value. If the function being mapped is a vector function, the callable must return a tuple of values instead. These extensions can be seen in the following code:

```
element = VectorElement("CG", cell, 1)
f = Function(element)
```

```
e = c*(f[0] + f[1])
def fh(x):
    return (x[0], x[1])
print e((0.5,0.7), { c: 10, f: fh })
```

To use expression evaluation for validating that the derivative computations are correct, spatial derivatives of form arguments can also be specified. The callable must then take a second argument which is called with a tuple of integers specifying the spatial directions in which to differentiate. A final example code computing $g^2 + g_{,0}^2 + g_{,1}^2$ for $g = x_0 x_1$ is shown below.

```
element = FiniteElement("CG", cell, 1)
g = Function(element)
e = g**2 + g.dx(0)**2 + g.dx(1)**2
def gh(x, der=()):
    if der == ():   return x[0]*x[1]
    if der == (0,): return x[1]
    if der == (1,): return x[0]
print e((2, 3), { g: gh })
```

19.8.6 Viewing expressions

Expressions can be formatted in various ways for inspection, which is particularly useful while debugging. The Python built in string conversion operator `str(e)` provides a compact human readable string. If you type `print e` in an interactive Python session, `str(e)` is shown. Another Python built in string operator is `repr(e)`. UFL implements `repr` correctly such that `e == eval(repr(e))` for any expression `e`. The string `repr(e)` reflects all the exact representation types used in an expression, and can therefore be useful for debugging. Another formatting function is `tree_format(e)`, which produces an indented multi-line string that shows the tree structure of an expression clearly, as opposed to `repr` which can return quite long and hard to read strings. Information about formatting of expressions as LaTeX and the dot graph visualization format can be found in the manual.

19.9 Implementation issues

19.9.1 Python as a basis for a domain specific language

Many of the implementation details detailed in this section are influenced by the initial choice of implementing UFL as an embedded language in Python.

Therefore some words about why Python is suitable for this, and why not, are appropriate here.

Python provides a simple syntax that is often said to be close to pseudo-code. This is a good starting point for a domain specific language. Object orientation and operator overloading is well supported, and this is fundamental to the design of UFL. The functional programming features of Python (such as generator expressions) are useful in the implementation of algorithms and form compilers. The built-in data structures `list`, `dict` and `set` play a central role in fast implementations of scalable algorithms.

There is one problem with operator overloading in Python, and that is the comparison operators. The problem stems from the fact that `__eq__` or `__cmp__` are used by the built-in data structures dict and set to compare keys, meaning that `a == b` must return a boolean value for `Expr` to be used as keys. The result is that `__eq__` can not be overloaded to return some `Expr` type representation such as `Equals(a, b)` for later processing by form compilers. The other problem is that `and` and `or` cannot be overloaded, and therefore cannot be used in `conditional` expressions. There are good reasons for these design choices in Python. This conflict is the reason for the somewhat non-intuitive design of the comparison operators in UFL.

19.9.2 Ensuring unique form signatures

The form compilers need to compute a unique signature of each form for use in a cache system to avoid recompilations. A convenient way to define a signature is using `repr(form)`, since the definition of this in Python is `eval(repr(form)) == form`. Therefore `__repr__` is implemented for all `Expr` subclasses.

Some forms are equivalent even though their representation is not exactly the same. UFL does not use a truly canonical form for its expressions, but takes some measures to ensure that trivially equivalent forms are recognized as such.

Some of the types in the `Expr` class hierarchy (subclasses of `Counted`), has a global counter to identify the order in which they were created. This counter is used by form arguments (both `BasisFunction` and `Function`) to identify their relative ordering in the argument list of the form. Other counted types are `Index` and `Label`, which only use the counter as a unique identifier. Algorithms are implemented for renumbering of all `Counted` types such that all counts start from 0.

In addition, some operator types such as `Sum` and `Product` maintains a sorted list of operands such that `a+b` and `b+a` are both represented as `Sum(a, b)`. The numbering of indices does not affect this ordering because a renumbering of the indices would lead to a new ordering which would lead to a different index renumbering if applied again. The operand sorting and renumbering combined ensure that the signature of equal forms will stay the same. To get the signature with renumbering applied, use `repr(form.form_data().form)`. Note that the representation, and thus the signature, of a form may change with versions of UFL.

19.9.3 Efficiency considerations

By writing UFL in Python, we clearly do not put peak performance as a first priority. If the form compilation process can blend into the application build process, the performance is sufficient. We do, however, care about scaling performance to handle complicated equations efficiently, and therefore about the asymptotic complexity of the algorithms we use.

To write clear and efficient algorithms in Python, it is important to use the built in data structures correctly. These data structures include in particular `list`, `dict` and `set`. CPython (Python programming language), the reference implementation of Python, implements the data structure `list` as an array, which means append, and pop, and random read or write access are all O(1) operations. Random insertion, however, is O(n). Both `dict` and `set` are implemented as hash maps, the latter simply with no value associated with the keys. In a hash map, random read, write, insertion and deletion of items are all $O(1)$ operations, as long as the key types implement `__hash__` and `__eq__` efficiently. Thus to enjoy efficient use of these containers, all `Expr` subclasses must implement these two special functions efficiently. The dict data structure is used extensively by the Python language, and therefore particular attention has been given to make it efficient (**?**).

19.10 Future directions

Many additional features can be introduced to UFL. Which features are added will depend on the needs of FEniCS users and developers. Some features can be implemented in UFL alone, while other features will require updates to other parts of the FEniCS project.

Improvements to finite element declarations is likely easy to do in UFL. The added complexity will mostly be in the form compilers. Among the current suggestions are space-time elements and related time derivatives, and enrichment of finite element spaces. Additional geometry mappings and finite element spaces with non-uniform cell types are also possible extensions.

Additional operators can be added to make the language more expressive. Some operators are easy to add because their implementation only affects a small part of the code. More compound operators that can be expressed using elementary operations is easy to add. Additional special functions are easy to add as well, as long as their derivatives are known. Other features may require more thorough design considerations, such as support for complex numbers which may affect many parts of the code.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code. However, the notion of metaprogramming and code generation adds another layer of abstraction which can make understanding the framework more difficult for end-users. Good error checks everywhere are therefore very important, to detect user errors as close as possible to the user input. The error messages, documentation, and unit test suite should be improved to help avoid frequently repeated errors and misunderstandings among new users.

Several algorithms in UFL can probably be optimized if bottlenecks are found as more complicated applications are attempted. The focus in the development has not been on achieving peak performance, which is not important in a tool like UFL.

To support form compiler improvements, algorithms and utilities for generating better code more efficiently can be implemented in UFL. In this area, more work on alternative automatic differentiation algorithms (**??**) can be useful. Another possibility for code improvement is operation scheduling, or reordering of the vertices of a graph partition to improve the efficiency of the generated code by better use of hardware cache and registers. Since modern C++ compilers are quite good at optimizing low level code, the focus should be on high level optimizations when considering potential code improvement in UFL and the form compilers. At the time of writing, operation scheduling is not implemented in UFL, and the value of implementing such an operation is an open question. However, results from (**?**) indicates that a high level scheduling algorithm could improve the efficiency of the generated code.

To summarize, UFL brings important improvements to the FEniCS framework: a richer form language, automatic differentiation and improved form compiler efficiency. These are useful features in rapid development of applications for efficiently solving partial differential equations. UFL improves upon the Automation of Discretization that has been the core feature of this framework, and adds Automation of Linearization. In conclusion, UFL brings FEniCS one step closer to its overall goal Automation of Mathematical Modeling.

19.11 Acknowledgements

This work has been supported by the Norwegian Research Council (grant 162730) and Simula Research Laboratory. I wish to thank everyone who has helped improving UFL with suggestions and testing, in particular Anders Logg, Kristian Ølgaard, Garth Wells, and Harish Narayanan. Both Kent-André Mardal and Marie Rognes performed critical reviews which greatly improved this manuscript.

CHAPTER 20

Unicorn: A Unified Continuum Mechanics Solver

By Johan Hoffman, Johan Jansson, Niclas Jansson and Murtazo Nazarov

Chapter ref: **[hoffman-2]**

20.1 Introduction

Unicorn is solver technology (models, methods, algorithms and software implementations) with the goal of automated simulation of realistic continuum mechanics applications, such as drag/lift computation for fixed or flexible objects (fluid-structure interaction) in turbulent incompressible or compressible flow (airplane/bird flight, car aerodynamics). The basis for Unicorn is Unified Continuum (UC) modeling formulated in Euler (laboratory) coordinates, together with a G2 (General Galerkin) adaptive stabilized FEM discretization with a moving mesh for tracking the phase interfaces. The UC model consists of canonical conservation equations for mass, momentum, energy and phase over the whole domain as one continuum, together with a Cauchy stress and phase variable as data for defining material properties and constitutive equation. Unicorn formulates and implements the adaptive G2 method applied to the UC model, and interfaces to other components in the FEniCS chain (FIAT, FFC, DOLFIN) providing representation of finite element function spaces, weak forms and mesh, and algorithms such as automated parallel assembly and linear algebra.

Unicorn as part of the FEniCS framework realizes automated computational modeling for general continuum mechanics applications in the form of canonical UC formulation in Euler coordinates, duality-based adaptive error control, tensor assembly, time-stepping, adaptive fixed-point iteration for solving discrete systems, mesh adaptivity by local cell operations (split, collapse, swap) (through MAdLib) and cell quality optimization (smoothing).

This chapter provides a description of the technology in Unicorn focusing on simple, efficient and general algorithms and software implementation of the UC concept and the adaptive G2 discretization. We describe how Unicorn fits into the FEniCS framework, how it interfaces to other FEniCS components and what interfaces and functionality Unicorn provides itself and how the implementation is designed. We also give use case application examples in fluid-structure interaction and adaptivity.

For a more detailed discussion on turbulence and adaptive error control in continuum mechanics we refer to chapter **??**.

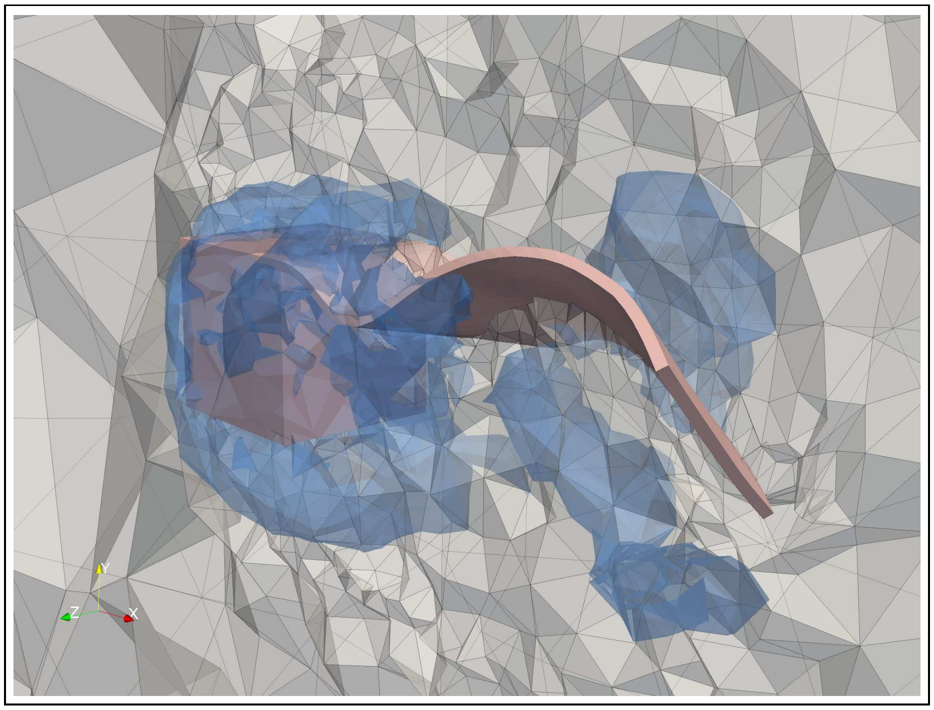


Figure 20.1: A fluid-structure example application of a flag mounted behind a cube in turbulent flow. The fluid-structure interface, an isosurface of the pressure and a cut of the mesh is plotted.

The Unicorn software is organized into three parts:

Library The Unicorn library publishes interfaces to and implements common solver technology such as automated time-stepping, error estimation/adaptivity, mesh smoothing/adaptation interface and slip/friction boundary condition.

Solver The Unicorn solver implements the G2 adaptive discretization method for the UC model by formulating the relevant weak forms and using the solver technology in the library and from other components of FEniCS. Currently there are two primary solvers: incompressible fluid/solid (including fluids-structure interaction) and compressible Euler (only fluid), where the long term goal is a unification of the incompressible/compressible formulations as well.

Applications Associated to the solver(s) are applications such as computational experiments/benchmarks with certain geometries, coefficients and parameters. These are represented as stand-alone programs built on top of the Unicorn solver/library, running in either serial or parallel (restricted to adaptive incompressible flow currently).

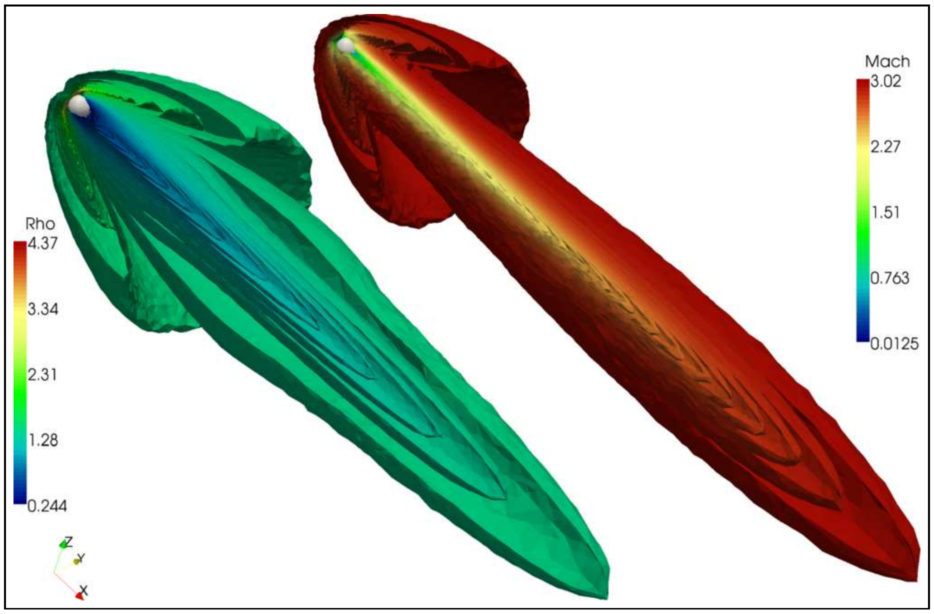


Figure 20.2: Example application of adaptive computation of 3D compressible flow around a sphere.

20.2 Acknowledgement

We acknowledge contributions to Unicorn, both software development as well as ideas and scientific support from: Mattias Aechtner, Peter Brune, Zilan Ciftci, Gäetan Compere, Niyazi Cem Degirmenci, Vida Heidarpour Dehkordi, Claes Johnson, Ashraful Kadir, Mohammad Reza Mokhtari, Jeannette Spühler, Michael Stöckli and Rodrigo Vilela de Abreu.

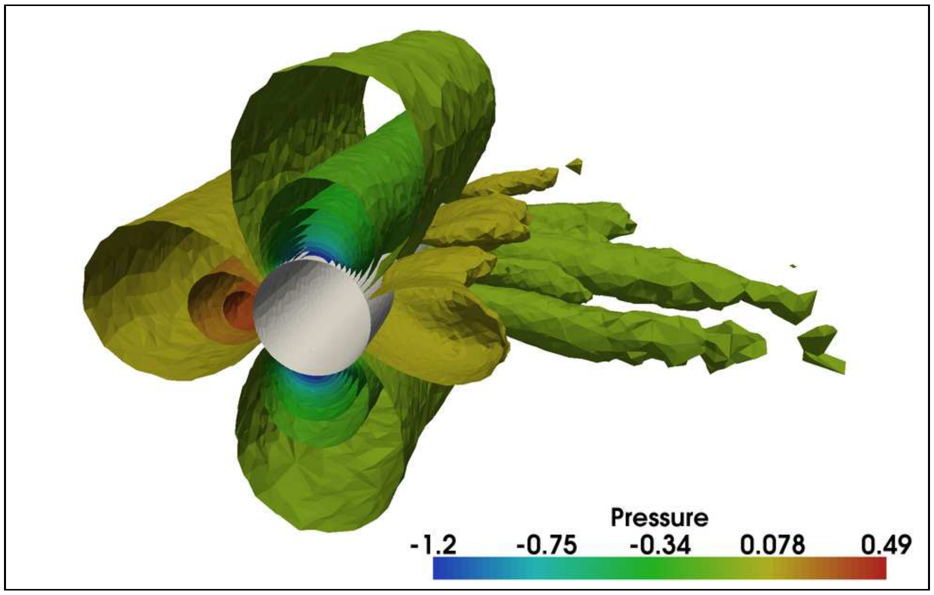


Figure 20.3: Example application of 3D turbulent incompressible flow around a cylinder with parallel adaptive computation.

20.3 Notation

We occasionally use an indexed Einstein notation with the derivative of a function f with regard to the variable x denoted as $D_x f$, and the derivative with regard to component x_i of component f_j denoted as $D_{x_i} f_j = \nabla f_j$. Repeated indices denote a sum: $D_{x_i} f_i = \sum_{i=1}^{d} D_{x_i} f_i = \nabla \cdot f$. Similarly we can express derivatives with respect to any variable: $D_u u = 1$.

20.4 Unified Continuum modeling

We define a unified continuum model in a fixed Euler coordinate system consisting of:

- conservation of mass
- conservation of momentum
- conservation of energy
- phase convection equation
- constitutive equations for stress as data

where the stress is the Cauchy (laboratory) stress and the phase variable is used to define material data such as constitutive equation for the stress and material parameters. Note that in this continuum description the coordinate system is fixed (Euler), and a phase function (marker) is convected according to the phase convection equation.

We define two variants of this model: incompressible and compressible, where a future aim is to construct a unified incompressible/compressible model and solver.

We start with a model for conservation of mass, momentum and energy, together with a convection equation for a phase function θ over a space-time domain $Q = \Omega \times [0, T]$ with Ω an open domain in R^3 with boundary Γ:

$$\boxed{\begin{aligned} D_t\rho + D_{x_j}(u_j\rho) &= 0 \quad \textbf{(Mass conservation)} \\ D_t m_i + D_{x_j}(u_j m_i) &= D_{x_j}\sigma_i \quad \textbf{(Momentum conservation)} \\ D_t e + D_{x_j}(u_j e) &= D_{x_j}\sigma_i u_i \quad \textbf{(Energy conservation)} \\ D_t\theta + D_{x_j}u_j\theta &= 0 \quad \textbf{(Phase convection equation)} \end{aligned}} \tag{20.1}$$

together with initial and boundary conditions. We can then pose constitutive relations between the constitutive (Cauchy) stress component σ and other variables such as the velocity u.

We define incompressibility as:

$$D_t\rho + u_j D_{x_j}\rho = 0$$

which together with mass and momentum conservation gives:

$$\begin{aligned} \rho(D_t u_i + u_j D_j u_i) &= D_{x_j}\sigma_{ij} \\ D_{x_j}u_j &= 0 \end{aligned}$$

where now the energy equation is decoupled and we can omit it.

We decompose the total stress into constitutive and forcing stresses:

$$D_{x_j}\sigma_{ij} = D_{x_j}\sigma_{ij} + D_{x_j}\sigma^f_{ij} = D_{x_j}\sigma_{ij} + f_i$$

Summarizing, we end up with the incompressible UC formulation:

$$\boxed{\begin{aligned} \rho(D_t u_i + u_j D_{x_j}u_i) &= D_{x_j}\sigma_{ij} + f_i \\ D_{x_j}u_j &= 0 \\ D_t\theta + D_{x_j}u_j\theta &= 0 \end{aligned}} \tag{20.2}$$

The UC modeling framework is simple and compact, close to the formulation of the original conservation equations, without mappings between coordinate systems. This allows simple manipulation and processing for error estimation and implementation. It is also general, we can choose the constitutive equations to model simple or complex solids and fluids, possibly both in interaction, with individual parameters.

20.4.1 Automated computational modeling and software design

One key design choice of UC modeling is to define the Cauchy stress σ as data, which means the conservation equations are fixed regardless of the choice of constitutive equation. This gives a generality in method and software design, where a modification of constitutive equation impacts the formulation/implementation of the constitutive equation, but not the formulation/implementation of the conservation equations.

In Unicorn we choose an Euler coordinate system and moving mesh, allowing a unified/canonical formulation of conservation equations. This enables automated computational modeling (with discretization and error control) for full fluid-structure (or multi-phase) problems, rather than to one component at a time, with unclear strategy how to combine them. We believe this is a unique method/software system in this respect.

20.5 Space-time General Galerkin discretization

The General Galerkin (G2) method has been developed as an adaptive stabilized finite element method for turbulent incompressible/compressible flow (Hoffman, 2005, 2006a,b, 2009, Hoffman and Johnson, 2006b, Nazarov, 2009) G2 has been shown to be cheap, since the adaptive mesh refinement is minimizing the number of degrees of freedom, general, since there are no model parameters to fit, and reliable, since the method is based on quantitative error control in a chosen output. The G2 method has been shown to accurately compute quantities of interest in both laminar and turbulent flow (Hoffman, 2005, 2006a,b, 2009, Hoffman and Johnson, 2006b, Hoffman et al., 2009).

We begin by describing the standard FEM applied to the model to establish basic notation, and proceed to describe streamline diffusion stabilization and local ALE map over a mesh T^h with mesh size h together with adaptive error control based on duality.

20.5.1 Standard Galerkin

We begin by formulating the standard cG(1)cG(1) FEM (Eriksson et al., 1996) with piecewise continuous linear solution in time and space for 28.5 by defining the exact solution: $w = [u, p, \theta]$, the discrete solution $W = [U, P, \Theta]$, the test function $v = [v^u, v^p, v^\theta]$ and the residual $R(W) = [R_u(W), R_p(W), R_\theta(W)]$:

$$
\begin{aligned}
R_u(W) &= \rho(D_t U_i + U_j D_{x_j} U_i) - D_{x_j}\Sigma_{ij} - f_i \\
R_p(W) &= D_{x_j} U_j \\
R_\theta(W) &= D_t\Theta + u_j D_{x_j}\Theta
\end{aligned}
$$

where $R(w) = 0$ and Σ denotes a discrete piecewise constant stress.

To determine the degrees of freedom ξ we enforce the Galerkin orthogonality $(R(W), v) = 0, \forall v \in V_h$ where v are test functions in the space of piecewise linear continuous functions in space and piecewise constant discontinuous functions in time and $(\cdot,\cdot)$ denotes the space-time L_2 inner product over Q. We thus have the weak formulation:

$$
\begin{aligned}
&(R^u(W), v^u) = (\rho(D_t U_i + U_j D_j U_i) - f_i, v_i^u) + (\Sigma_{ij}, D_{x_j} v_i^u) - \int_{t_{n-1}}^{t_n}\int_\Gamma \Sigma_{ij} v_i^u n_j \, ds dt = 0 \\
&(R^p(W), v^p) = (D_{x_j} U_j, v^p) = 0 \\
&(R^\theta(W), v^\theta) = (D_t\Theta + u_j D_{x_j}\Theta, v^\theta) = 0
\end{aligned}
$$

for all $v \in V_h$, where the boundary term on Γ arising from integration by parts vanishes if we assume a homogenous Neumann boundary condition for the stress Σ.

This standard finite element formulation is unstable for convection-dominated problems and due to choosing equal order for the pressure and velocity. Thus we cannot use the standard finite element formulation by itself but proceed to a streamline diffusion stabilization formulation. We also describe a local ALE discretization for handling the phase interface.

20.5.2 Local ALE

If the phase function Θ has different values on the same cell it would lead to an undesirable diffusion of the phase interface. By introducing a local ALE coordinate map (Eriksson et al., 1996) on each discrete space-time slab based on a given mesh velocity (i.e. the material velocity of one of the phases) we can define the phase interface at cell facets, allowing the interface to stay discontinuous. We describe the details of the coordinate map and its influence on the FEM discretization in (Hoffman et al., 2009). The resulting discrete phase equation is:

$$D_t\Theta(x) + (U(x) - \beta_h(x)) \cdot \nabla\Theta(x) = 0 \tag{20.3}$$

with $\beta_h(x)$ the mesh velocity.

We thus choose the mesh velocity β_h to be the discrete material velocity U in the structure part of the mesh (vertices touching structure cells) and in the rest of the mesh we use mesh smoothing to determine β_h to maximize the mesh quality according to a chosen objective, alternatively use local mesh modification operations (refinement, coarsening, swapping) on the mesh to maintain the quality (Compère et al., 2009). Note that we still compute in Euler coordinates, but with a moving mesh.

20.5.3 Streamline diffusion stabilization

For the standard FEM formulation of the model we only have stability of U but not of spatial derivatives of U. This means the solution can be oscillatory, causing inefficiency by introducing unnecessary error. We instead choose a weighted standard Galerkin/streamline diffusion method of the form $(R(W), v + \delta R(v)) = 0, \forall v \in V_h$ (see (Eriksson et al., 1996)) with $\delta > 0$ a stabilization parameter. We here also make a simplification where we only introduce necessary stabilization terms and drop terms not contributing to stabilization. Although not fully consistent, the streamline diffusion stabilization avoid unnecessary smearing of shear layers as the stabilization is not based on large ($\approx h^{-\frac{1}{2}}$) cross flow derivatives). For the UC model the stabilized method thus looks like:

$$\begin{aligned}
(R^u(W), v^u) &= (\rho(D_t U_i + U_j D_j U_i) - f_i, v_i^u) + (\Sigma_{ij}, D_{x_j} v_i^u) + SD^u(W, v^u) = 0 \\
(R^p(W), v^p) &= (D_{x_j} U_j, v^p) + SD^p(W, v^p) = 0
\end{aligned}$$

for all $v \in V_h$, and:

$$\begin{aligned}
SD^u(W, v^u) &= \delta_1(U_j D_j U_i, U_j^u D_j v_i^u) + \delta_2(D_{x_j} U_j, D_{x_j} v_j^u) \\
SD^p(W, v^p) &= \delta_1(D_{x_i} P, D_{x_i} v^p)
\end{aligned}$$

where we only include the dominating stabilization terms to reduce complexity in the formulation.

20.5.4 Duality-based adaptive error control

We give a summary for illustrative purposes of the mathematical framework for duality based error estimation in the context of the UC formulation.

We consider an equation (such as the UC) in residual form $R(u) = 0$. We define a linearized operator satisfying: $A(\bar{U})e = R(U)$. We can then construct an error estimate as follows, where ψ is the quantity of interest.

It is of critical importance in science and engineering to be able to bound the computational error $e = u - U$, we can phrase this as requiring that e satisfies:

$$|(e, \psi)| < TOL \qquad (20.4)$$

We define:

$$\begin{aligned} (Aw, v) = (w, A^* v) \quad &\text{(adjoint definition)} \\ A^*\phi = \psi \quad &\text{(dual equation)} \end{aligned}$$

We can then express the error in terms of the residual and dual solution:

$$(e, \psi) = (e, A^*\phi) = (Ae, \phi) = (-R(U), \phi) \quad \text{(error representation)}$$

We note we could alternatively start from the weak formulation:

$$(e, \psi) = a^*(\phi, e) = a(e, \phi) = (-R(U), \phi)$$

We thus have a canonical formula for the error estimate:

$$|(e, \psi)| \leq \sum_K \|hR(U)\|_K \|D\phi\|_K = \sum_K \mathcal{E}_K \quad \text{(error bound)}$$

where $\mathcal{E}_K$ is an error indicator on each cell K.

We can thus define an adaptive algorithm for controlling $|(e, \psi)|$ by making sure that U satisfies:

$$\begin{gathered} \sum_K \mathcal{E}_K < TOL \\ \Rightarrow \\ |(e, \psi)| < TOL \end{gathered}$$

by iteratively reducing h in cells with large $\mathcal{E}_K$ until the tolerance condition is satisfied.

20.5.5 Unicorn/FEniCS software implementation

We implement the G2 discretization of the UC (including adaptive error control) in a general interface for time-dependent PDE where we give the forms $a(U, v) = (D_U F_U, v)$ and $L(v) = (F_U, v)$ for assembling the linear system given by Newton's method for a time step for the incompressible UC with Newtonian fluid constitutive equation in figure 20.4. The language used is the FEniCS Form Compiler (FFC) (Kirby and Logg, 2006) form notation. In the adaptive error control, we similarly represent residuals as forms in the form language, which then allows us to compute error indicators using basic programming constructs.

An overview of the interfaces and implementations of the Unicorn data structures and algorithms are presented below.

```
...

def ugradu(u, v):
    return [dot(u, grad(v[i])) for i in range(d)]

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

def S(u, P):
    return mult(P, Identity(d)) - mult(nu, grad(u))

def f(u, v):
    return -dot(ugradu(Uc, Uc), v) + \
        dot(S(Uc, P), grad(v)) + \
        -mult(d1, dot(ugradu(Um, u), ugradu(Um, v))) + \
        -mult(d2, dot(div(u), div(v))) + \
        dot(ff, v)

def dfdu(u, k, v):
    return -dot(ugradu(Um, u), v) + \
        -dot(mult(nu, grad(u)), grad(v)) + \
        -mult(d1, dot(ugradu(Um, u), ugradu(Um, v))) + \
        -mult(d2, dot(div(u), div(v)))

# cG(1)
def F(u, u0, k, v):
    uc = 0.5 * (u + u0)
    return (-dot(u, v) + dot(u0, v) + mult(k, f(u, v)))

def dFdu(u, u0, k, v):
    uc = 0.5 * u
    return (-dot(u, v) + mult(1.0 * k, dfdu(uc, k, v)))

a = (dFdu(U1, U0, k, v)) * dx
L = -F(UP, U0, k, v) * dx
```

Figure 20.4: Source code for bilinear and linear forms for incompressible UC one time step with a Newton-type method (approximation of Jacobian).

20.6 Unicorn solver

The Unicorn solver: `UCSolver` ties together the technology in the Unicorn library with other parts of FEniCS to expose an interface (see listing 20.5) for simulating applications in continuum mechanics. The main part of the solver implementation is the weak forms for the G2 discretization of the UC model (see listing 20.4), together with forms for the stress and residuals for the error estimation. Coefficients from the application are connected to the form, and then time-stepping is carried out by `TimeDependentPDE`. Certain coefficients, such as the δ stabilization coefficients are also computed as part of the solver (not as forms), where we aim to compute them as part of the forms. The solver computes one iteration of the adaptive algorithm (primal solve, dual solve and mesh refinement), where the adaptive loop is implemented by iteratively running the solver for a sequence of meshes.

The `UCSolver` implementation is parallel, where we can show scaling on hundreds of processes on a BlueGene L (see (Jansson, 2008)). The entire adaptive algorithm is parallel (including Rivara mesh refinement) and Unicorn can simulate massively parallel applications for turbulent incompressible flow. Mesh smoothing and adaptivity are not yet enabled in parallel (and thus no fluid-structure interaction is possible) but this is work in progress.

A compressible variant of the `UCSolver` exists as the `CNSSolver` for adaptive G2 for compressible Euler flow. The general method and algorithm is very close to that of the `UCSolver`, aside from the incompressibility. The long term goal is a unification of the incompressible/compressible formulations as well. We refer to (Nazarov, 2009) for implementation details of the compressible `CNSSolver`.

There also exists variants of the UC solver as part of the Unicorn source code representing older formulations of the method, or pure incompressible fluid formulations. These are used to verify that the general `UCSolver` does not introduce performance overhead or implementation errors.

20.7 Unicorn library classes: data types and algorithms

20.7.1 Unicorn software design

We can view an adaptive simulator of continuum mechanics as the root of a tree where each node is a technology (method/algorithm/implementation) with edges representing dependencies. Unicorn implements the levels of the tree closest to the root (application at the root, solver, library above), below Unicorn are other components of FEniCS and other libraries such as MAdLib. If the technology at the root is to produce a meaningful simulation, every technology at each node in the tree needs to fit into the theoretical framework, be free of critical bugs, be reasonably efficient and keep a stable interface for its parents for maintainability. If any of the nodes fail in any of these perspectives, the simulator at the root will likely not run at all or consist of garbage and be meaningless.

Since software engineering is a discipline where more manpower may not increase productivity, but can actually decrease it (Brooks, 1995), there is a critical need to control the complexity in the tree. To achieve this, Unicorn follows two basic design principles:

```
/// Unicorn G2 method implementation for the incompressible UC model
class UCSolver : public TimeDependentPDE, public MeshAdaptInterface
{
public:
  /// Constructor: give boundary conditions, coefficients, parameters
  UCSolver(Function& U, Function& U0, Function** bisect, Mesh& mesh,
           Array <BoundaryCondition*>& bc_mom,
           Array <BoundaryCondition*>& bc_con,
           Function** f,
           real T, real nu, real mu, real rho_f, real rho_s,
           real u_bar, TimeDependent& t, PDEData* pdedata);

  /// Prescribe mesh size for MeshAdaptInterface
  virtual void updateSizeField();

  /// Allocate/deallocate PDE data for dynamic mesh adaptivity (MeshAdaptInterface)
  virtual void allocateAndComputeData();
  virtual void deallocateData();

  /// Compute mesh vertex coordinates and velocity
  void computeX();
  void computeW();

  /// Compute density, pressure, stress
  void computeRho();
  void computeP();
  void computeStress();

  // Compute initial theta by either a given xml file or a bisect function
  void computeTheta0();

  /// From TimeDependentPDE: time-stepping control
  void shift();
  void preparestep();
  bool update(real t, bool end);

  /// From TimeDependentPDE: computeW, Um, Wm, Stabilization, computeP, computeB
  void prepareiteration();

  /// Assemble time step residual (L)/ right hand side of Newton
  void rhs(const Vector& x, Vector& dotx, real T);

  /// Compute initial value
  void u0(Vector& x);

  /// Save solution/output quantities
  void save(Function& U, real t);

  /// Compute least-squares stabilization parameter (delta)
  void computeStabilization(Mesh& mesh, Function& w,
                            real nu, real k, real t,
                            Vector& d1vector, Vector& d2vector);

  /// Deform/move mesh
  void deform(Mesh& mesh, Function& W, Function& W0);

  /// Smooth/optimize quality of all or part of the mesh
  void smoothMesh(bool bAdaptive);
}
```

Figure 20.5: C++ class interface for the Unicorn solver: UCSolver.

```
(v1, v2, v3) = TestFunctions(TH)  # test basis function
(rho, m, e) = TrialFunctions(TH)  # trial basis function
(rho0, m0, e0) = Functions(TH)    # solution from previous time step
...
# S prefix denotes stabilization: a1_a is Galerkin, S1_a is stabilization

# Bilinear form for density
a1_a = v1*rho*dx - k*0.5*dot(grad(v1),u)*rho*dx + \
     k*0.5*v1*rho*udotnormal*ds
S1_a = k*0.5*delta*dot(grad(v1),U)*dot(U, grad(rho))*dx + \
    k*0.5*nu_rho*dot(grad(v1),grad(rho))*dx

# Linear form for the density
a1_L = v1*rho0*dx + k*0.5*dot(grad(v1),u)*rho0*dx - \
    k*0.5*v1*rho0*udotnormal*ds
S1_L = - k*0.5*delta*dot(grad(v1),U)*dot(U, grad(rho0))*dx - \
    k*0.5*nu_rho*dot(grad(v1),grad(rho0))*dx

a2_a, S2_a, a2_L, S2_L = 0, 0, 0, 0

for i in range(0, d):
    # Bilinear form for momentum m_i
    a2_a += v2[i]*m[i]*dx - k*0.5*dot(grad(v2[i]),u)*m[i]*dx + \
        k*0.5*v2[i]*m[i]*udotnormal*ds
    S2_a += k*0.5*delta*dot(grad(v2[i]),U)*dot(U, grad(m[i]))*dx + \
        k*0.5*nu_m[i]*dot(grad(v2[i]),grad(m[i]))*dx

    # Linear form for the momentum
    a2_L += v2[i]*m0[i]*dx + k*0.5*dot(grad(v2[i]),u)*m0[i]*dx + \
        k*v2[i].dx(i)*P*dx - k*0.5*v2[i]*m0[i]*udotnormal*ds
    S2_L += -k*0.5*delta*dot(grad(v2[i]),U)*dot(U, grad(m0[i]))*dx - \
        k*0.5*nu_m[i]*dot(grad(v2[i]),grad(m0[i]))*dx

# Bilinear form for energy
a3_a = v3*e*dx - k*0.5*dot(grad(v3),u)*e*dx + k*0.5*v3*e*udotnormal*ds
S3_a = k*0.5*delta*dot(grad(v3),U)*dot(U, grad(e))*dx + \
    k*0.5*nu_e*dot(grad(v3),grad(e))*dx

# Linear form for energy
a3_L = v3*e0*dx + k*0.5*dot(grad(v3),u)*e0*dx + k*dot(grad(v3),u)*P*dx - \
    k*0.5*v3*e0*udotnormal*ds
S3_L = - k*0.5*delta*dot(grad(v3),U)*dot(U,grad(e0))*dx - \
    k*0.5*nu_e*dot(grad(v3),grad(e0))*dx

# Weak form of G2 for the Euler equations:
a = a1_a + S1_a + a2_a + S2_a + a3_a + S3_a
L = a1_L + S1_L + a2_L + S2_L + a3_L + S3_L
```

Figure 20.6: Source code for bilinear and linear forms for G2 for compressible Euler one time step with a Picard fixed-point iteration.

Prioritize simplicity of interfaces and implementation
We can rephrase this as the KISS principle: Keep It Simple and Stupid

Avoid premature optimization
"Premature optimization is the root of all evil" (Donald Knuth)]

Together, these two principles enforce generality and understandability of interfaces and implementations. Unicorn re-uses other existing implementations and chooses straightforward, sufficiently efficient (optimize bottlenecks) standard algorithms for solving problems. This leads to small, simple and maintainable implementations. High performance is achieved by reducing the computational load on the method level (adaptivity and fixed-point iteration).

20.7.2 Unicorn classes/interfaces

Unicorn consists of key concepts abstracted in the following classes/interfaces:

`TimeDependentPDE`**: time-stepping**
In each time-step a non-linear algebraic system is solved by fixed-point iteration.

`ErrorEstimate`**: adaptive error control**
The adaptive algorithm is based on computing local *error indicators* of the form $\epsilon_K = \|hR(U)\|_K \|D\phi\|_K$.

`SpaceTimeFunction`**: space-time coefficient**
Storage and evaluation of a space-time function/coefficient.

`SlipBC`**: friction boundary condition**
Efficient computation of turbulent flow in Unicorn is based on modeling of turbulent boundary layers by a friction model, where the normal condition: $u \cdot n = 0, x \in \Gamma$, is implemented as a strong boundary condition in the algebraic system.

`ElasticSmoother`**: elastic mesh smoothing/optimization**
Optimization of cell quality according to an elastic analogy.

`MeshAdaptInterface`**: mesh adaptation interface**
Abstraction of the interface to the MAdLib package for mesh adaptation using local mesh operations.

20.7.3 `TimeDependentPDE`

We consider time-dependent equations of the type $f(u) = -D_t u + g(u) = 0$ where g can include differential operators in space, where specifically the UC model is of this type. In weak form the equation type looks like$(f(u), v) = (-D_t u + g(u), v) = 0$, possibly with partial integration of terms

We want to define a class (datatype and algorithms) abstracting the time-stepping of the G2 method, where we want to give the equation (possibly in weak form) as input and generate the time-stepping automatically. cG(1)cG(1) (Crank-Nicolson-type discretization in time) gives the equation for the (possibly non-linear) algebraic system $F(U)$ (in Python notation):

```
# cG(1)
def F(u, u0, k, v):
    uc = 0.5 * (u + u0)
    return (-dot(u, v) + dot(u0, v) + mult(k, g(uc, v)))
```

With v: $\forall v \in V_h$ generating the equation system.

We solve this system by Newton-type fixed-point iteration:

$$(F'(U_P)U_1, v) = (F'(U_P) - F(U_P), v) \tag{20.5}$$

where U_P denotes the value in the previous iterate and $F' = D_U F$ the Jacobian matrix or an approximation. Note that the choice of F' only affects the convergence of the fixed-point iteration, and does not introduce approximation error.

We define the bilinear form $a(U, v)$ and linear form $L(v)$ corresponding to the left and right hand sides respectively (in Python notation):

```
def dFdu(u, u0, k, v):
    uc = 0.5 * u
    return (-dot(u, v) + mult(k, dgdu(uc, k, v)))

a = (dFdu(U, U0, k, v)) * dx
L = (dFdu(UP, U0, k, v) - F(UP, U0, k, v)) * dx
```

Thus, in each time step we need to solve the system given in eq. 20.5 by fixed-point iteration by repeatedly computing a and L, solving a linear system and updating U.

We now encapsulate this in a C++ class interface in fig. 20.7 which we call `TimeDependentPDE` where we give a and L, an end time T, a mesh (defining V_h) and boundary conditions.

The skeleton of the time-stepping with fixed-point iteration is implemented in listing 20.7.3.

To simplify the discussion, we consider Newton's method as a linearization of the continuum model, where we then compute the discretization of each successive iteration. A more general formulation would be to compute Newton's method of the discretization. For many cases this would be equivalent formulations, since $D_U(F(U), v) = (D_U F(U), v)$, but for a stabilized method these formulations may not be the same.

We use a block-diagonal Newton method, where we start by formulating the full Newton method and then drop terms. We also use the constitutive law as an identity to remove instability caused by iterating between σ and U. We formulate Newton's method for the system $F(X) = (F_u(X), F_\sigma(X), F_p(X))^\top = 0$, with $X = (U, \sigma, P)^\top$, where the three components are decoupled.

See (Jansson, 2009) for a dicussion about the efficiency of the fixed-point iteration and its implementation.

```
/// Represent and solve time dependent PDE.
class TimeDependentPDE
{
/// Public interface
public:
  TimeDependentPDE(
    // Computational mesh
    Mesh& mesh,
    // Bilinear form for Jacobian approx.
    Form& a,
    // Linear form for time-step residual
    Form& L,
    // List of boundary conditions
    Array <BoundaryCondition*>& bcs,
    // End time
    real T);
  /// Solve PDE
  virtual uint solve();

protected:
  /// Compute initial value
  virtual void u0(Vector& u);
  /// Called before each time step
  virtual void preparestep();
  /// Called before each fixed-point iteration
  virtual void prepareiteration();
  /// Return the bilinear form a
  Form& a();
  /// Return the linear form L
  Form& L();
  /// Return the mesh
  Mesh& mesh();
};
```

Figure 20.7: C++ class interface for TimeDependentPDE.

20.7.4 ErrorEstimate

The duality-based adaptive error control algorithm requires the following primitives:

Residual computation We compute the mean-value in each cell of the continuous residual $R(U) = f(U) = -D_tU + g(U)$, this is computed as the L_2-projection into the space of piecewise constants W_h: $(R(U), v) = (-D_tU + g(U), v), \forall v \in W_h$ (the mean value in each cell) as a form representing the continuous residual.

Dual solution We compute the solution of the dual problem using the same technology as the primal problem. The dual problem is solved backward

```
void TimeDependentPDE::solve()
{
  // Time-stepping
  while(t < T)
  {
    U = U0;
    preparestep();
    step();
  }
}

void TimeDependentPDE::step()
{
  // Fixed-point iteration
  for(int iter = 0; iter < maxiter; iter++)
  {
    prepareiteration();
    step_residual = iter();

    if(step_residual < tol)
    {
      // Iteration converged
      break;
    }
  }
}

void TimeDependentPDE::iter()
{
  // Compute one fixed-point iteration
  assemble(J, a());
  assemble(b, L());
  for (uint i = 0; i < bc().size(); i++)
    bc()[i]->apply(J, b, a());
  solve(J, x, b);

  // Compute residual for the time-step/fixed-point equation
  J.mult(x, residual);
  residual -= b;

  return residual.norm(linf);
}
```

Figure 20.8: Skeleton implementation in Unicorn of time-stepping with fixed-point iteration.

in time, but with the time coordinate transform $s = T - t$ we can use the standard `TimeDependentPDE` interface and step the dual time s forward.

Space-time function storage/evaluation We compute error indicators while solving the dual problem as space-time integrals over cells: $\epsilon_K = (R(U), D_x\Phi)_{L_2(K\times T)}$, where we need to evaluate both the primal solution U and the dual solution Φ. In addition, U is a coefficient in the dual equation. This requires storage and evaluation of a space-time function, which is encapsulated in the `SpaceTimeFunction` class.

Mesh adaptation After the computation of the error indicators we select the largest $p\%$ of the indicators for refinement. The refinement is then performed by recursive Rivara cell bisection or by general mesh adaptation using Madlib (Compère et al., 2009), which is based on edge split, collapse and swap operations, and thus gives the ability to coarsen a mesh, or more generally to control the mesh size.

Using these primitives, we can construct an adaptive algorithm. The adaptive algorithm is encapsulated in the C++ class interface in fig. 20.9 which we call `ErrorEstimate`.

```
/// Estimate error as local error indicators based on duality
class ErrorEstimate
{
public:

  /// Constructor (give components of UC residual and dual solution)
  ErrorEstimate(Mesh& mesh,
                Form* Lres_1, Form* Lres_2, Form* Lres_3,
                Form* LDphi_1, Form* LDphi_2, Form* LDphi_3);

  // Compute error (norm estimate)
  void ComputeError(real& error);

  // Compute error indicator
  void ComputeErrorIndicator(real t, real k, real T;

  // Compute largest indicators
  void ComputeLargestIndicators(std::vector<int>& cells,
                                real percentage);

  // Refine based on indicators
  void AdaptiveRefinement(real percentage);
}
```

Figure 20.9: C++ class interface for `ErrorEstimate`.

20.7.5 SpaceTimeFunction

The error estimation algorithm requires, as part of solving the dual problem, the evaluation of space-time coefficients (at time s in the dual problem we need to evaluate the primal solution U at time t = T - s). This requires storage and evaluation of a space-time function, which is encapsulated in the SpaceTimeFunction class (see listing 20.10).

The space-time functionality is implemented as a list of space functions at regular sample times, where evaluation is piecewise linear interpolation in time of the values in the dofs.

```
/// Representation of space−time function (storage and evaluation)
class SpaceTimeFunction
{
public:

  /// Create space−time function
  SpaceTimeFunction(Mesh& mesh, Function& Ut);

  /// Evaluate function at time t, giving result in Ut
  void eval(real t);

  // Add a space function at time t
  void addPoint(std::string Uname, real t);

  /// Return mesh associated with function
  Mesh& mesh();

  /// Return interpolant function
  Function& evaluant();
```

Figure 20.10: C++ class interface for SpaceTimeFunction.

20.7.6 SlipBC

For high Reynolds numbers problems such as car aerodynamics or airplane flight, it's not possible to resolve the turbulent boundary layer. One possibility is to model turbulent boundary layers by a friction model:

$$u \cdot n = 0 \tag{20.6}$$

$$u \cdot \tau_k + \beta^{-1} n^\top \sigma \tau_k = 0, k = 1, 2 \tag{20.7}$$

We implement the normal component condition (slip) boundary condition strongly. By "strongly" we here mean an implementation of the boundary condition after assembling the left hand side matrix and the right hand side vector in the algebraic system, whereas the tangential components (friction) are implemented "weakly" by adding boundary integrals in the variational formulation. The row of the matrix and load vector corresponding to a degree of freedom is found and replaced by a new row according to the boundary condition.

The idea is as follows: Initially, the test function v is expressed in the Cartesian standard basis (e_1, e_2, e_3). Now, the test function is mapped locally to normal-tangent coordinates with the basis (n, τ_1, τ_2), where $n = (n_1, n_2, n_3)$ is the normal, and $\tau_1 = (\tau_{11}, \tau_{12}, \tau_{13})$, $\tau_2 = (\tau_{21}, \tau_{22}, \tau_{23})$ are tangents to each node on the boundary. This allows us to let the normal direction to be constrained and the tangent directions be free:

$$v = (v \cdot n)n + (v \cdot \tau_1)\tau_1 + (v \cdot \tau_2)\tau_2.$$

For the matrix and vector this means that the rows corresponding to the boundary need to be multiplied with n, τ_1, τ_2, respectively, and then the normal component of the velocity should be set to 0.

This concept is encapsulated in the class `SlipBC` which is a subclass of `dolfin::BoundaryCondition` for representing strong boundary conditions. The implementation is based on multiplying elements of the matrix with components of n, τ_1, τ_2.

For more details about the implementation of slip boundary conditions we refer to (Nazarov, 2009).

20.7.7 `ElasticSmoother`

```
def tomatrix(q):
    return [ [q[d * i + j] for i in range(d)] for j in range(d) ]

Fmatrix = tomatrix(F)
Fm = tomatrix(F)
F0m = tomatrix(F0)
vm = tomatrix(v)

# icv is inverse cell volume

def f(U, F, v):
    return (-dot(mult(Fm, grad(U)), vm) - \
     dot(mult(transp(grad(U)), Fm), vm)))

a = (dot(dotF, v)) * dx
L = (mult(icv, dot(F0m, vm) + mult(k, f(U, F, vm)))) * dx
```

Figure 20.11: Source code for forms representing one time step for the deformation gradient (F) evolution in the elastic smoother variant of the UC model.

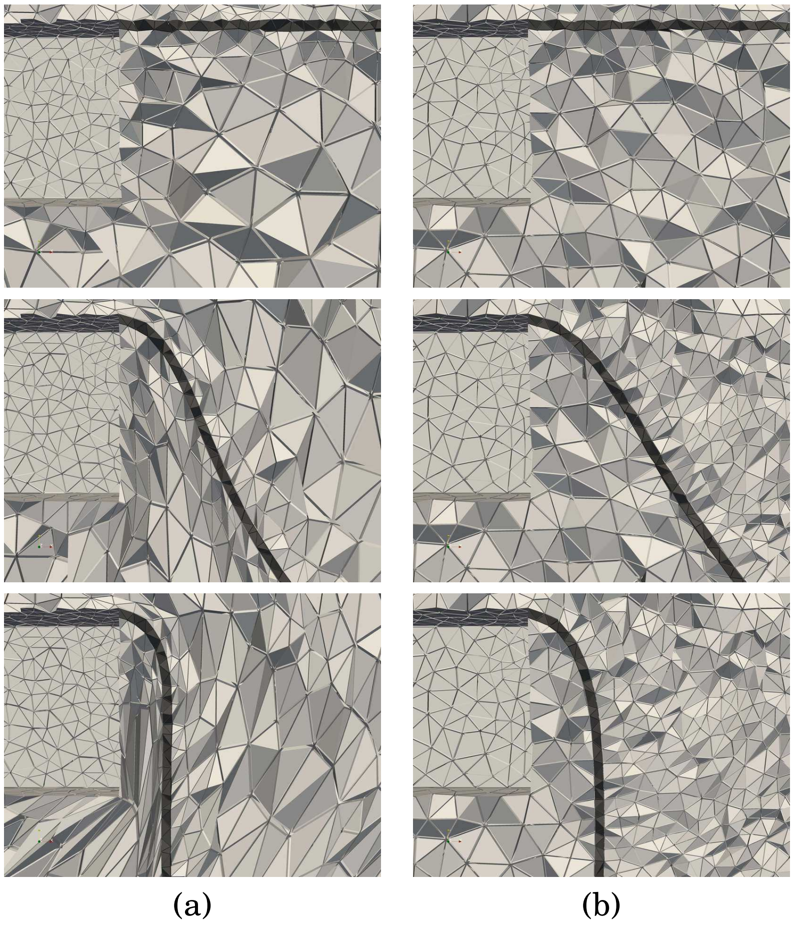


Figure 20.12: Robustness test with (a) elastic smoothing and (b) mesh adaptation. Note the badly shaped cells squeezed between the cube and flag.

To maintain a discontinuous phase interface in the UC with fluid-structure data, we define the mesh velocity β_h as the discrete velocity U in the solid phase (specifically on the interface). The mesh velocity in the fluid can be chosen more arbitrarily, but has to satisfy mesh quality and size criteria. We construct a cell quality optimization/smoothing method based on a pure elastic variant of the UC.

We define the following requirements for the mesh velocity β_h:

1. $\beta_h = U$ in the solid phase part of the mesh.

2. Bounded mesh quality Q in the fluid part of the mesh. Preferably the mesh smoothing should improve Q if possible.

3. Maintain mesh size $h(x)$ close to $\hat{h}(x)$ given by a posteriori error estimation in an adaptive algorithm.

```
/// Optimize cell quality according to elastic variant of UC model
class ElasticSmoother
{
public:

  ElasticSmoother(Mesh& mesh);

  /// Smooth smoothed_cells giving mesh velocity W over time step k
  /// with h0 the prescribed cell size
  void smooth(MeshFunction<bool>& smoothed_cells,
              MeshFunction<bool>& masked_cells,
              MeshFunction<real>& h0,
              Function& W, real k);

  /// Extract submesh (for smoothing only marked cells)
  static void submesh(Mesh& mesh, Mesh& sub,
                      MeshFunction<bool>& smoothed_cells,
                      MeshFunction<int>& old2new_vertex,
                      MeshFunction<int>& old2new_cell);

}
```

Figure 20.13: C++ class interface for `ElasticSmoother`.

We formulate a simplistic variant of the UC model where we only consider a solid, and we omit the incompressibility equation (see listing 20.11). We use a constitutive law $\sigma = \mu(I - (FF^\top)^{-1})$ where we recall F as the deformation gradient. We use the update law: $D_t F^{-1} = -F^{-1}\nabla u$ where we thus need an initial condition for F. We set the initial condition $F_0 = \bar{F}$ where $\bar{F}$ is the deformation gradient with regard to a scaled equilateral reference cell, representing the optimal shape with quality $Q = 1$.

Solving the elastic model can thus be seen as optimizing for the highest global quality Q in the mesh. We also introduce a weight on the Young's modulus μ for cells with low quality, penalizing high average, but low local quality over mediocre global quality. We refer to the source code for more details.

As an alternative to mesh smoothing we can consider using local mesh modification operations (refinement, coarsening, swapping) on the mesh to maintain the quality (Compère et al., 2009) through `MeshAdaptInterface`.

Unicorn provides the `ElasticSmoother` class (see listing 20.13, which can be used to smooth/optimize for quality all or part of the mesh.

We perform a robustness test of the elastic smoothing and the mesh adaptivity shown in 20.12 where we use the same geometry as the turbulent 3D flag problem, but define 0 inflow velocity and instead add a gravity body force to the flag to create a very large deformation with the flag pointing straight down. Both the elastic smoothing and the mesh adaptvity compute solutions, but as expected, the elastic mesh smoothing eventually cannot control the cell quality (there does not exist a mesh motion which can handle large rigid body rotations while bounding the cell quality).

20.7.8 `MeshAdaptInterface`

A critical component in the adaptive algorithm as described above is *Mesh adaptivity*, which we define as constructing a mesh satisfying a given mesh size function $h(x)$.

We start by presenting the Rivara recursive bisection algorithm (Rivara, 1992) as a basic choice for mesh adaptivity (currently the only available choice for parallel mesh adaptivity), but which can only refine and not coarsen. Then the more general MAdLib is presented, which enables the full mesh adaptation to the prescribed $h(x)$ through local mesh operations: edge split, edge collapse and edge swap.

Rivara recursive bisection

The Rivara algorithm bisects (splits) the longest edge of a cell, thus replacing the cell with two new cells, and uses recursive bisection to eliminate non-conforming cells with hanging nodes. A non-conforming cell K_1 has a neighbor (incident) cell K_2 that has a vertex on an edge of cell K_1.

Algorithm 9 The Rivara recursive bisection algorithm

```
procedure BISECT($K$)
    Split longest edge $e$
    while $K_i(e)$ is non-conforming do
        BISECT($K_i$)
    end while
end procedure
```

The same algorithm holds in both 2D/3D (triangles/tetrahedra). In 2D, it can be shown (Rivara, 1992) that the algorithm terminates in a finite number of steps, and that the minimum angle of the refined mesh is at least half the minimum angle of the starting mesh. In practice the algorithm produces excellent quality refined meshes both in 2D and 3D.

Local mesh operations: Madlib

Madlib incorporates an algorithm and implementation of **mesh adaptation** where a small set of local mesh modification operators are defined such as edge split, edge collapse and edge swap. A mesh adaptation algorithm is defined which uses this set of local operators in a control loop to satisfy a prescribed size field

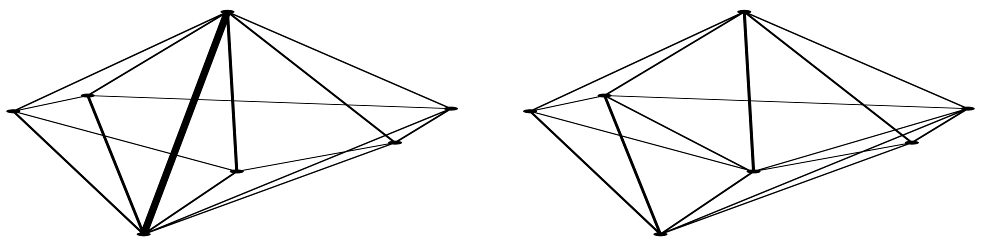


Figure 20.14: Edge swap operation: (a) initial cavity with swap edge highlighted (b) possible configuration after the swap.

$h(x)$ and quality tolerance. Edge swapping is the key operator for improving quality of cells, for example around a vertex with a large number of connected edges.

In the formulation of finite element methods it is typically assumed that the cell size of a computational mesh can be freely modified to satisfy a desired size field $h(x)$ or to allow mesh motion. In state-of-the-art finite element software implementations this is seldom the case, where typically only limited operations are allowed (Bangerth et al., 2007, COMSOL, 2009), (local mesh refinement), or a separate often complex, closed and ad-hoc mesh generation implementation is used to re-generate meshes.

The mesh adaptation algorithm in Madlib gives the freedom to adapt to a specified size field using local mesh operations. The implementation is published as free software/open source allowing other research to build on the results and scientific repeatability of numerical experiments.

Unicorn provides the `MeshAdaptInterface` class (see listing 20.15, where one can subclass and implement virtual functions to control the mesh adaptation.

We perform a robustness test of the elastic smoothing and the mesh adaptivity shown in 20.12, see a more detailed description in the elastic smoothing section.

20.8 Solving continuum mechanics problems

In this section we give use cases for modeling and solving continuum mechanics problems using the Unicorn technology. We start with a use case for solving a fluid-structure problem without adaptivity, where we cover modeling of geometry and subdomains, coefficients, dynamic allocation of PDE data for mesh adaptivity and specification of main program (interface to running the solver). Next, we present a use case for solving a turbulent pure fluid problem with adaptivity, where we cover modeling of data for the dual problem, the adaptive loop, and specifying slip/friction boundary conditions for modeling turbulent boundary layers.

```
  /// Interface to MAdLib for mesh adaptation using local operations
  /// Subclass and implement the virtual functions
  class MeshAdaptInterface
  {
  public:
    MeshAdaptInterface(Mesh *);

  protected:
    /// Start mesh adaptation algorithm
    void adaptMesh();

    /// Give cell size field
    virtual void updateSizeField() = 0;

    /// Allocate and deallocate solver data
    virtual void deallocateData() = 0;
    virtual void allocateAndComputeData() = 0;

    /// Constrain entities not to be adapted
    void constrainExternalBoundaries();
    void constrainInternalBoundaries();

    /// Add functions to be automatically interpolated
    void addFunction(string name, Function** f);
    void clearFunctions();
  };
}
```

Figure 20.15: C++ class interface for `MeshAdaptInterface`.

20.8.1 Fluid-structure

We here give a use case of solving a fluid-structure continuum mechanics problem, where the user specifies data for modeling the problem, and illustrates interfaces and expected outcomes. We divide the use case into 4 parts:

Geometry and subdomains
The user specifies possible geometrical parameters and defines subdomains. We note that for complex geometries the user may omit geometry information and specify subdomain markers as data files.

Coefficients
Known coefficients such as a force function and boundary conditions are declared.

PDE data
The user subclasses a PDEData class and specifies how the PDE data is

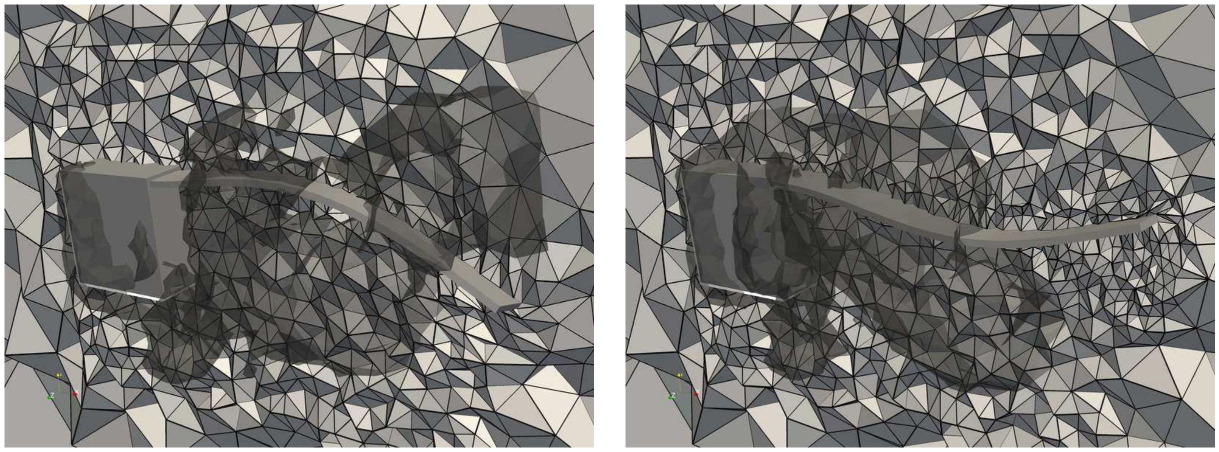


Figure 20.16: Snapshot of flag simulation FSI use case with (a) elastic smoothing and (b) dynamic mesh adaptation. A cut of the mesh is shown together with an isosurface of the pressure to visualize the flow.

constructed and destroyed. This construction/destruction may happen during the simulation if the mesh is adapted.

main program
The user implements the main program and declares and passes data to to the solver.

20.8.2 Adaptivity

We continue with a use case for adaptive solution of a pure fluid turbulent flow problem: flow around a 3D cylinder. The implementation of the problem is very similar to the fluid-structure case (just with pure fluid data), but with 3 important additions:

Dual problem
To compute the error estimate required by the adaptive algorithm, we must solve a dual problem generated by the primal problem and an output quantity ψ. Since the dual problem is similar in form to the primal problem, we implement both as variants of the same solver.

In this case we are interested in computing drag, which gives ψ as a boundary condition for the dual problem:

```
CylinderBoundary cb;
SubSystem xcomp(0);
Function minus_one(mesh, -1.0);
```

```
DirichletBC dual_bc0(minus_one, mesh, cb, xcomp);

Array <BoundaryCondition*> dual_bc_mom;
dual_bc_mom.push_back(&dual_bc0);
```

Adaptive loop

We construct the program to compute one iteration of the adaptive loop: solve primal problem, solve dual problem, compute error estimate and check if tolerance is satisfied, compute adapted mesh. We can then run the adaptive loop simply by a loop which runs the program (here in Python which we also use to move data according to iteration number):

```
offset = 0
N = 20

for i in range(offset, N):
    dirname = ``iter_%2.2d'' % i
    mkdir(dirname)

    system(``./unicorn-cylinder > log'')
    for file in glob(``./*.vtu''):
    move(file, dirname)
    for file in glob(``./*.pvd''):
    move(file, dirname)
```

Slip boundary condition

For turbulent flow we model the boundary layer as a friction boundary condition. We specify the normal component as a string slip boundary condition used just as a regular Dirichlet boundary condition. The `xcomp` variable denotes an offset for the first velocity component in a system (for compressible Euler the system is [density, velocity, energy], and we would thus give component 2 as offset).

```
SlipBoundary sb;
SubSystem xcomp(0);

SlipBC slip_bc(mesh, sb, xcomp);

Array <BoundaryCondition*> primal_bc_mom;
primal_bc_mom.push_back(&slip_bc);
```

```
#include <dolfin.h>
#include <unicorn/FSIPDE.h>

using namespace dolfin;
using namespace dolfin::unicorn;

real bmarg = 1.0e-3 + DOLFIN_EPS;

namespace Geo
{
  // Geometry details ///////////////////////////////////////////////////
  real box_L = 3.0;
  real box_H = 2.0;
  real box_W = 2.0;

  real xmin = 0.0; real xmax = box_L;
  real ymin = 0.0; real ymax = box_H;
  real zmin = 0.0; real zmax = box_W;
}

// Sub domain for inflow
class InflowBoundary3D : public SubDomain
{
public:
  bool inside(const real* p, bool on_boundary) const
  {
    return on_boundary && (p[0] < Geo::xmax - bmarg);
  }
};

// Sub domain for outflow
class OutflowBoundary3D : public SubDomain
{
public:
  bool inside(const real* p, bool on_boundary) const
  {
    return on_boundary && (p[0] > Geo::xmax - bmarg);
  }
};
```

Figure 20.17: Part 1 of Unicorn solver FSI use case: geometry and subdomains.

```
// Force term
class ForceFunction : public Function
{
public:
  ForceFunction(Mesh& mesh, TimeDependent& td) : Function(mesh), td(td) {}
  void eval(real* values, const real* x) const
  {
    int d = cell().dim();

    for(int i = 0; i < d; i++)
    {
      values[i] = 0.0;
    }
  }

  TimeDependent& td;
};

// Boundary condition for momentum equation
class BC_Momentum_3D : public Function
{
public:
  BC_Momentum_3D(Mesh& mesh, TimeDependent& td) :
    Function(mesh), td(td) {}
  void eval(real* values, const real* x) const
  {
    int d = cell().dim();

    for(int i = 0; i < d; i++)
    {
      values[i] = 0.0;
    }
    if (x[0] < (Geo::xmin + bmarg))
      values[0] = 100.0;
  }

  TimeDependent& td;
};


// Initial condition for phase variable
class BisectionFunction : public Function
{
public:
  BisectionFunction(Mesh& mesh) : Function(mesh) {}
  void eval(real* values, const real* p) const
  {
    // NB: We specify the phase variable as xml data so
    // this function is not used

    bool condition = true;

    if (condition)
      values[0] = 0.0;
    else
      values[0] = 1.0;
  }
```

```
class FlagData : public PDEData
{
public:
  void create(Mesh& mesh)
  {
    bcf_mom = new BC_Momentum_3D(mesh, td);
    bcf_con = new BC_Continuity_3D(mesh);
    f = new ForceFunction(mesh, td);

    bisect = new BisectionFunction(mesh);
    zero = new Function(mesh, 0.0);

    bc_mom0 = new DirichletBC(*bcf_mom, mesh,
                              iboundary);
    bc_con0 = new DirichletBC(*bcf_con, mesh,
                              oboundary);

    bc_mom.clear();
    bc_con.clear();

    bc_mom.push_back(bc_mom0);
    bc_con.push_back(bc_con0);
  }

  void destroy()
  {
    delete bcf_mom;
    delete bcf_con;
    delete f;
    delete bisect;
    delete zero;

    delete bc_mom0;
    delete bc_con0;
  }

  Function* bcf_mom;
  Function* bcf_con;
  Function* f;
  Function* bisect;
  Function* zero;

  DirichletBC* bc_mom0;
  DirichletBC* bc_con0;

  Array <BoundaryCondition*> bc_mom;
  Array <BoundaryCondition*> bc_con;

  InflowBoundary3D iboundary;
  OutflowBoundary3D oboundary;

  TimeDependent td;
};
```

Figure 20.19: Part 3 of Unicorn solver FSI use case: problem data.

```
int main()
{
  Mesh mesh("flag.xml");

  real nu = 0.0;
  real nus = 0.5;
  real rhof = 1.0;
  real rhos = 1.0;

  real E = 1.0e6;

  real T = 0.2;

  dolfin::set("ODE␣number␣of␣samples", 500);

  Function U, U0;

  real u_bar = 100.0;

  FlagData pdedata;

  ICNSPDE pde(U, U0, &(pdedata.bisect), mesh,
              pdedata.bc_mom, pdedata.bc_con,
              &(pdedata.f), T, nu, E, nus, rhof, rhos,
              u_bar, pdedata.td, &pdedata);

  // Compute solution
  pde.solve(U, U0);

  return 0;
}
```

Figure 20.20: Part 4 of FSI use case: main program, passing data to solver.

CHAPTER 21

Viper: A Minimalistic Scientific Plotter

By Ola Skavhaug

Chapter ref: **[skavhaug]**

CHAPTER 22

Lessons Learned in Mixed Language Programming Using Python, C++ and SWIG

By Johan Hake and Kent-Andre Mardal

Chapter ref: **[mixedlanguage]**

22.1 Introduction

Python (Python programming language) has in the last decade become an established platform for scientific computing. Widely used scientific software like, e.g., PETSc(PETSc software package), HYPRE(Hypre), Trilinos(Sala et al., 2008), VTK(VTK softwar
ITK, GiNaC(Bauer et al., 2000) have all been equipped with Python interfaces. In addition many packages like for instance the FEniCSpackages FErari, FIAT(Kirby, 2006), FFC(**?**), UFL(**?**), Viper, as well as other packages like SymPy(**?**), SciPy(**?**) are pure Python packages. The DOLFINlibrary has both a C++ and a Python user-interface. Python makes application building on top of DOLFINmore user friendly, but the Python interface also introduces additional complexity and new problems. This chapter describes some lessons learned during the development of PyDOLFIN, and is intentionally quite technical. We assume that the reader has basic knowledge of both C++ and Python. A suitable textbook on Python for scientific computing is (**?**), which cover both Python and its C interface. SWIG is well documented and we refer to the user manual that can be found on its web page (**?**). Finally, we refer to **?** and Sala et al. (2008) for a description of how SWIGcan be used to generate Python interfaces for Diffpack and Trilinos.

All the code examples in this chapter can found in `$FENICSBOOK/mixed_language/`.

22.2 Using SWIG

Python and C++ are two very different languages, but they can be glued together as specified by the Python C-API (**?**), also see **?**. The code that glues together C++ and Python is commonly called wrapper code. Writing wrapper code manually is often cumbersome, therefore wrapper code generators such as e.g. F2PY (**?**), SIP (**?**), Siloon (**?**), or SWIG (**?**) are usually used. Common for all these code generators are that they create a Python extension module in the form of a shared library. This module can then be imported and accessed as any Python module. SWIGhas been used to create PyDOLFIN. SWIGis a mature wrapper code generator that support many languages and is extensively documented.

22.2.1 Basic SWIG

To get a basic understanding of SWIG, consider the following definition of an Array class defined in `Array.h`.

```
#include <iostream>

class Array {
public:
  // Constructors and destructors
  Array(int n_=0);
  Array(int n_, double* a_);
  Array(const Array& a_);
  ~Array();

  // Operators
  Array& operator=(const Array& a_);
  const double& operator [] (int i) const;
  double& operator [] (int i);
  const Array& operator+= (const Array& b);

  // Methods
  int dim() const;
  double norm() const;

private:
  int n;
  double *a;

};

std::ostream & operator<< ( std::ostream& os, const Array& a);
```

A first attempt to make the Array accessible in Python using SWIG, is to write a SWIGinterface file `Array_1.i`.

```
%module Array
%{
#include "Array.h"
%}
%include "Array.h"
```

Here we specify the name of the Python module: `Array`, what code should be inlined directly in the wrapper code (declarations), `#include "Array.h"` and what code SWIGshould parse to create the wrapper code `%include "Array.h"` (definitions). The following command shows how to run SWIGon this interface file to produce the wrapper code.

```
swig -python -c++ -I. -O Array_1.i
```

The command generates two files: `Array.py` and `Array_wrap.cxx`. In the file `Array_wrap.cxx`, SWIGincludes everything that is needed to bridge the interface between our C++ class and Python. When `Array_wrap.cxx` is compiled into a shared library it can be imported directly into Python, which is done in the generated `Array.py` file. The file `Array.py` is pure python, and in this file SWIGhas generated code for the so called Python proxy class; a Python version of the C++ defined `Array` class. The reader should be able to recognize the Python class `Array` in the end of the `Array.py` file.

The following Distutils file executes the SWIGcommand above and compiles and links the source code and the generated wrapper code into a shared library.

```
import os
import numpy
from distutils.core import setup, Extension
swig_cmd ='swig -o Array_wrap.cxx -python -c++ -O -I. Array_1.i'
os.system(swig_cmd)
sources = ['Array.cpp','Array_wrap.cxx']
setup(name = 'Array',
      py_modules = ["Array"],
      ext_modules = [Extension('_' + 'Array', sources, \
                   include_dirs=['.', numpy.get_include() + "/numpy"])])
```

Build and install the module in the current working directory by typing:

```
python setup.py install --install-lib=.
```

The Python proxy class resembles the C++ class in many ways. Simple methods like `dim()` and `norm()` will be wrapped correctly to Python.[1] However, there exists a number of corner cases that SWIGdoes not map correctly:

[1]This is because SWIGmaps int and double arguments to the corresponding Python types through built-in typemaps. SWIGalso uses typemaps between objects of declared types, like our `Array` class, meaning that the copy constructor will work as expected.

1. the `operator[]` does not work,
2. the `operator+=` returns a new Python object (with different `id`),
3. printing does not use the `std::ostream & operator<<`,
4. the `Array(int n_, double* a_);` constructor is not working properly.

Hence, a number of different problems arise even in such a simple example. Fortunately, these problems are fairly common and simple, and general solutions to these problems can be implemented quite easily. We will go through each of these issues.

22.2.2 The `operator[]`

The first problem is that SWIGdoes not wrap the `operator[]` from C++ to Python. One basic observation here is that Python does not have 'const' types. Hence, the `operator[]` would in this case be ambiguous[2]. In Python the operator should map to two different special methods: `__setitem__` and `__getitem__`. To implement this operator properly, we ignore both version of the `operator[]` with

```
%ignore Array::operator[];
```

and then tell SWIGto extend the C++ extension layer of `Array` with the mentioned methods. This is done using the `%extend` directive.

```
%extend Array {
double __getitem__(int i) {
  return (*self)[i];
}

void __setitem__(int i, double v) {
  (*self)[i] = v;
}
...
};
```

Notice that all SWIGdirectives start with '%'. Furthermore, the access to the actual instance is provided by the `self` pointer, which in this case is a C++ pointer that points to an `Array` instance. The pointer is comparable to the `this` pointer in a C++ class, but with only public attributes available. SWIGalso extends the python proxy class with the same methods.

[2]In general will SWIGsolve such ambiguities by ignoring one of the methods while issuing a warning.

22.2.3 `operator +=` and memory

The second problem is related to SWIGthat garbage collection in Python. Python features garbage collection, which means that a user should not be bothered with the destruction of objects. The mechanism is based on reference counting. When no more references are pointing to an object it is destroyed. The SWIGgenerated Python module consists of a small Python layer, that defines the interface to the underlying C++ object. An instance of a SWIGgenerated class therefore keeps a reference to the underlying C++ object. Default behavior is that the C++ object is destroyed together with the Python object. This behavior can be troublesome in quite many cases, which can be illustrated with the following simple example:

```
from Array import Array
def add(b):
    print "id(b):",id(b)
    b+=b
    print "id(b):",id(b)

a = Array(10)
print "id(a):",id(a)
add(a)
a+=a
```

This script produce the following output:

```
id(a): 3085535980
id(b): 3085535980
id(b): 3085536492
Segmentation fault
```

The script causes a segmentation fault because the underlying C++ object is destroyed after the call to `add()`. When the last `a+=a` is performed there are no C++ object to be added. This happens because the SWIGgenerated `__iadd__` method returns a new Python object, which will be destroyed in our example. This is illustrated by the different result from the `id()` function[3]. The two calls to `id(b)` return different numbers, which means that a new Python object is returned by the SWIGgenerated `__iadd__` method. The variable `b` is local in the `add` function. Before the call to `__iadd__`, `b` will point to the same Python object as `a` does; they have the same `id`. After the call will `b` point to a new Python object, which is local for the `add` function. When we leave the function, the reference count to `b` is decreased to zero and `b` will be destroyed, together with the underlying C++ object. Therefore, when `__iadd__` is called at the end of the script, `a`'s underlying C++ object has been deleted and we get the segmentation fault.

[3]Taking `id` of a Python object returns a unique reference to that object.

This problem is solved by extending the C++ extension layer with an _add method, which use the operator+= directly. Furthermore, we extend the Python proxy class with our own __iadd__ method that use the _add function. The following code snippet illustrates the use of %extend:

```
%extend Array {
...
  void _add(const Array& a){
    (*self) += a;
  }

  %pythoncode %{
    def __iadd__(self,a):
      self._add(a)
      return self
  %}
...
};
```

The same script will report the same id for all objects after the suggested changes has been applied. No objects are created or deleted and we avoid the segmentation fault.

22.2.4 std::ostream & operator<<

SWIGignores the operator <<, and this operator is therefore useless from Python. However, we can again use the %extend directive to make this operator available from Python. We do this by extending the C++ extension layer with a __str__ method.

```
\begin{code}
%extend Array {
...
  std::string __str__() {
    std::ostringstream s;
    s << (*self);
    return s.str();
  }
};
```

This method use the operator<< to pipe the stream representation of array to a std::ostringstream and then return a std::string representation of the stream. To make SWIGable to convert a std::string to a Python string, we need to include the std_string.i file in the Array_2.i file. In Python we can then call print on an instance of Array, with the result of displaying the content of the Array.

22.2.5 The constructor: Array(int n_, double* a_);

The fourth problem is related to pointer handling in C/C++ and how SWIGdeals with pointers. From the constructor signature alone, it is not clear whether double * points to a single value or to the first element of an array. Therefore, SWIGtakes a conservative approach and handles pointers as pointers, not assuming anything about the length. In this example, we do of course know that double* a points to the first element of an array of length n. However, SWIGprovides the concept 'typemap' to enable mappings between C/C++ types and Python objects. The following code demonstrates how to map a Numpy array to, e.g., the (int n_, double* a_) constructor.

```
%typemap(in) (int n_, double* a_){
  if (!PyArray_Check($input)) {
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array");
    return NULL; ;
  }
  PyArrayObject* pyarray = reinterpret_cast<PyArrayObject*>($input);
  if (!(PyArray_TYPE(pyarray) == NPY_DOUBLE)) {
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array of doubles");
    return NULL; ;
  }
  $1 = PyArray_DIM(pyarray,0);
  $2 = static_cast<double*>(PyArray_DATA(pyarray));
}
```

A reader not familiar with the C-APIs of Python and NumPywill probably consider this typemap code as fairly technical, but it is a good example as it demonstrates some of the possibilities with typemaps.

The first line specifies that the typemap should be applied to input (in) arguments to the C++ library, which has the signature int n_,double* a_. The $ prefixed variables are used to map in and output variables in the typemap. The variables $1 and $2 maps to the first and second argument of the typemap, i.e., n_ and a_. Furthermore, $input maps to a pointer to the Python object a user is calling the SWIGgenerated method with.

In the next three lines we check if the input Python object is a NumPyarray, and raise an exception if not. Note that any Python C-API function that returns NULL tells the Python interpreter that an exception has occurred. Python will then raise the error set by the PyErr_SetString statement. Next, we cast the

Python object pointer to a NumPyarray pointer and check if the data type of the NumPyarray is correct, i.e. that it contains doubles. Then, we acquire the data from the NumPyarray and assign the two input variables.

A user defined typemap should be followed by a `%typecheck` directive in case of overloading. SWIGrely on such directives to resolve which of the overloaded C++ methods that should be called when the corresponding Python method is called[4]. If a wrapped C++ class has overloaded methods, SWIGdynamically needs to figure out which one of them it should call. This process is called dynamic dispatch. SWIGfirst check the number of arguments. If several methods has the same number of arguments SWIGuse a priority system, based on an internal type priority numbering. See the SWIGdocumentation (**?**) for more information on the built in type priorities SWIGuses. When a user define a typemap for a new type he also need to associate the typemap with such a priority number, which is done by the `%typecheck` directive.

A suitable typecheck for our example typemap looks like:

```
%typecheck(SWIG_TYPECHECK_DOUBLE_ARRAY) (int n_, double* a_) {
   $1 = PyArray_Check($input) ? 1 : 0;
 }
```

Here `SWIG_TYPECHECK_DOUBLE_ARRAY` is a `typedef` for the priority number assigned for arrays of doubles. The typecheck should return a 1 if the Python object `$input` has the correct type, and 0 otherwise.

22.3 SWIGand PyDOLFIN

We are now ready to describe some of the specializations we have done in an effort to make PyDOLFINboth usable and more *Pythonic*. The interface files resides in the `dolfin/swig` directory, and are organized into *i)* global files, which applies to the whole DOLFINlibrary, and *ii)* kernel module files that applies to specific modules in DOLFIN. The latter files are divided into ..._`pre.i` and ..._`post.i` files, which are applied respectively before and after the inclusion of the header files of the particular kernel module. The modules follows the catalog structure of DOLFIN: `common`, `parameters`, `la`, `mesh` and so forth. The global interface files are all included in `dolfin.i`, the main SWIGinterface file. The kernel module interface files are included together with the C++ header files, in the automatically generated `kernel_modules.i` file.

We will here walk through the main interface file of `dolfin.i` and address the global interface files. Then we will address some issues in the module specific interface files.

[4]In Python you cannot overload class methods, i.e., only one method with the same name per class is allowed. You can define several methods with the same name in Python, however, only one of them will actually exist.

22.3.1 `dolfin.i` and the `cpp` module

The file `dolfin.i` starts by defining the name of the generated Python module.

```
%module(package="dolfin", directors="1") cpp
```

This statement tells SWIGto create a module called `cpp` that resides in the package of `dolfin`. We have also enabled the use of directors. The latter is required to be able to subclass DOLFINclasses in Python. We will return to this below. By naming the generated extension module `cpp`, and putting it in the `dolfin` Python package, we hide the generated interface into a sub module of dolfin; the `dolfin.cpp` module. A user can access the `cpp` module from Python by:

```
import dolfin.cpp as cpp
```

In the `dolfin` module we then import the generated classes and functions we want to expose to the `dolfin` namespace. This is done in the `__init__.py` file that resides in the `site-packages/dolfin/` directory. In `__init__.py` we also import pure Python classes and functions, which are defined in Python module files. These files also reside in the `site-packages/dolfin/` directory.

The next two blocks in `dolfin.i` defines code that will be inserted into the SWIGgenerated C++ wrapper file.

```
%{
#include <dolfin/dolfin.h>
#define PY_ARRAY_UNIQUE_SYMBOL PyDOLFIN
#include <numpy/arrayobject.h>
%}

%init%{
import_array();
%}
```

SWIGwill insert any code that resides in a %{...}% block, verbatim at the top of the generated C++ wrapper file (%{...}% is short for `%header`%{...}%). Hence, the first block of code is similar to the include statements you would put in a standard C++ program. The code in the second block, `%init`%{...}%, is inserted in the code for the Python module initialization. The `import_array()` function is needed to initialize the C-API of NumPy. SWIGprovides several such blocks, each inserting verbatim code into the wrapper file at different positions, see SWIGdocumentation for more alternatives(**?**).

22.3.2 Reference counting using shared_ptr

In the example with the wrapping of the `operator+=`[5] method above, we see that it is important to prevent premature destruction of the underlying C++ object. A nice feature of SWIGis that we can declare that a wrapped class shall store the underlying C++ object using a `shared_ptr` instead of a raw pointer. By doing this SWIGdoes not have to explicitly delete the C++ object when the reference count of the Python object reach zero, but rather decrease the count on the `shared_ptr`. DOLFINprovides a `shared_ptr` interface for some crucial classes, which interact nicely with the `shared_ptr` stored C++ objects in DOLFIN.

To get this working in PyDOLFINwe need to include the `boost_shared_ptr.i` file. This file declares two user macros: `SWIG_SHARED_PTR` and `SWIG_SHARED_PTR_-DERIVED`. These macros needs to be called for the classes we want to use `shared_ptr` for. In PyDOLFINwe do this in the `shared_ptr_classes.i` file. In addition to store instance of the particular class using a `shared_ptr`, the macros also declares typemaps for passing a `shared_ptr` stored object to a method that expects a reference or pointer to such an objects. This means that the typemap pass a dereferenced `shared_ptr` to the function. This behavior can lead to unintentional trouble as we circumvent the `shared_ptr` mechanism.

In DOLFINwe store instances of some crucial classes internally with `shared_ptr`s. The same classes are naturally declared as being stored with `shared_ptr` in Python, using the above mention directives. When objects of these classes are passed as argument to methods or constructors in DOLFIN, we usually define two such methods: a `shared_ptr` and a reference version. The following code snippet illustrate two constructors of `Function`, which each takes a `FunctionSpace` as an argument [6]:

```
/// Create function on given function space
explicit Function(const FunctionSpace& V);

/// Create function on given function space (shared data)
explicit Function(boost::shared_ptr<const FunctionSpace> V);
```

As instances of `FunctionSpace` in PyDOLFINis stored using `shared_ptr` we want SWIGto use the second constructor. However, SWIGgenerates de-reference typemaps for the first constructor. So when we instantiate a `Function` with a `FunctionSpace`, SWIGwill unfortunately pick the first constructor instead of the correct second one. The consequences for this is that the `FunctionSpace` is passed without increasing the reference count of the `shared_ptr`. This undermines the whole concept of `shared_ptr`. To prevent this faulty behavior we ignore the reference constructor from the interface that is wrapped (see `function_pre.i`).

[5]A discussion of how to implement operator+= in C++ can be found in (**?**).

[6]Instances of `FunctionSpace` are internally stored using `shared_ptr`.

```
%ignore dolfin::Function::Function(const FunctionSpace&);
```

22.3.3 Typemaps

The types included in the `kernel_module.i` file are mostly wrapped nicely with SWIG. However, as in the `Array` example above, there exists corner-cases which are problematic. In `dolfin.i` we include three different types of global typemaps: *i)* general-, *ii)* NumPy- and, *iii)* std_vector-typemaps. These are implemented in the interface files: `typemaps.i`, `numpy_typemaps.i` and `std_vector_typemaps.i`. We will here present some of the typemaps defined in these files.

`typemaps.i`:

In `typemaps.i` we define typemaps for four different basic types. In- and out-typemaps for `dolfin::uint`, and `dolfin::real`, an in-typemap for `int`, and an out-typemap macro for `std::pair<dolfin::uint,dolfin::uint>`.

We start with the simplest typemap, an out-typemap for `dolfin::uint` (notice that Python does not have `unsigned int`):

```
%typemap(out) dolfin::uint = int;
```

This typemap specifies that a function returning a `dolfin::uint` should use the built-in out-typemap for `int`. Hence, SWIGlet us reuse a typemap simply by copying it. We could have used the same feature for the corresponding in-typemap, however an unfortunate bug force us to implement the whole typemap from scratch. The typemap looks like this:

```
%typemap(in) dolfin::uint
{
  if (PyInteger_Check($input))
  {
    long tmp = static_cast<long>(PyInt_AsLong($input));
    if (tmp>=0)
      $1 = static_cast<dolfin::uint>(tmp);
    else
      SWIG_exception(SWIG_TypeError, "expected positive 'int' for argumen
  }
  else
    SWIG_exception(SWIG_TypeError, "expected positive 'int' for argument
}
```

We see that the typemap resembles the NumPytypemap above. We first check that the object is of integer type. The check is performed by the `PyInteger_Check` function. We have implemented the `PyInteger_Check` function instead of using the built in Python C-API macro `PyInt_Check`, which combined with NumPy, cause the above mentioned bug. Next, we then convert the Python integer to a `long` and check if it is positive. Finally, we assign the input argument $1 to a `dolfin::uint` casted version of the value. If one of the checks fails we use a built in SWIGfunction, `SWIG_exception` to raise a python exception. These predefined SWIGexceptions are defined in the `exception.i` file, which we need to include in our `dolfin.i` file. The `$argnum` variable expands to the argument number of a function or methods that expects a `dolfin::uint`. Including this variable in the string will create a more understandable error message. Finally we also define a corresponding typecheck for the typemap, which is not shown here. After the `uint` typemap we also define an in-typemap for the `int` type, which is almost a copy of the `uint` typemap and therefore not presented here.

The out-typemap for `std::pair<dolfin::uint,dolfin::uint>` returns a Python tuple of two integers:

```
%typemap(out) std::pair<dolfin::uint,dolfin::uint>
{
  $result = Py_Build Value("ii",$1.first,$1.second);
}
```

This is an example of a short and comprehensive typemap. It uses the Python C-API function `Py_BuildValue` to build a tuple of the two values in the `std::pair` object.

`numpy_typemaps.i:`

In `numpy_typemaps.i` we define in-typemaps for arrays of primitive types: `double`, `int` and `dolfin::uint`. As in the `Array` example above, we define in-typemaps for these types so one can pass a NumPyarray of the corresponding type as the argument. Instead of writing one typemap for each primitive type, we write a SWIGmacro, which is called using the different types as argument. The code in the typemaps are inserted directly in the wrapper code, with the different variable names $1 $2, `$input`, substituted with the actual argument names. This can produce a lot of code as some of these typemaps are used frequently. We have therefore put the typemap code into a function, which is called from the typemap instead. The whole macro looks like:

```
%define UNSAFE_NUMPY_TYPEMAPS(TYPE,TYPE_UPPER,NUMPY_TYPE,TYPE_NAME,DESCR)
%{
SWIGINTERN bool convert_numpy_to_ ## TYPE_NAME ## _array_no_check(PyObjec
{
  if PyArray_Check(input)
  {
    PyArrayObject *xa = reinterpret_cast<PyArrayObject*>(input);
    if ( PyArray_TYPE(xa) == NUMPY_TYPE )
    {
      ret  = static_cast<TYPE*>(PyArray_DATA(xa));
      return true;
    }
  }
  PyErr_SetString(PyExc_TypeError,"numpy array of 'TYPE_NAME' expected. M
  return false;
}
%}

%typecheck(SWIG_TYPECHECK_ ## TYPE_UPPER ## _ARRAY) TYPE *
{
    $1 = PyArray_Check($input) ? 1 : 0;
}

%typemap(in) TYPE *
{
if (!convert_numpy_to_ ## TYPE_NAME ## _array_no_check($input,$1))
    return NULL;
}

%apply TYPE* {TYPE* _array}
%enddef
```

The first line defines the signature of the macro. The macro is called using 5 arguments:

- `TYPE`: The name of the primitive type: `dolfin::uint`, `double`
- `TYPE_CHECK`: The name of the corresponding typecheck-name SWIGuses: `INT32`, `DOUBLE`
- `NUMPY_TYPE`: The name of the NumPytype: `NPY_UINT`, `NPY_DOUBLE`
- `TYPE_NAME`: The short typename: `uint`, `double`
- `DESCR`: A description character used in NumPyto describe the type: `'I'`, `'d'`

We can then call the macro to instantiate the typemaps and typechecks.

```
UNSAFE_NUMPY_TYPEMAPS(dolfin::uint,INT32,NPY_UINT,uint,I)
UNSAFE_NUMPY_TYPEMAPS(double,DOUBLE,NPY_DOUBLE,double,d)
```

Here we have instantiated the typemap for a `dolfin::uint` and a `double` array. The typemap does not use any check of the length of the handed NumPyarray. This means that a user can easily trigger a segmentation fault, and it is why we have named the typemap unsafe.

The typemap function

```
  SWIGINTERN bool convert_numpy_to_ ## TYPE_NAME ## _array_no_check(PyObj
```

takes a pointer to a `PyObject` as input. The function will return `true` if the conversion is successful and `false` otherwise. The converted array will be returned by the `TYPE*& ret` argument. The peculiar naming convention of `to_ ## TYPE_NAME ## _array` will be translated into `to_double_array` if `TYPE_NAME` is set to `double`

The `%apply TYPE* {TYPE* _array}` directive means that we want the typemap to apply to any argument of type `TYPE*` with argument name `_array`. This is another way of copying a typemap, similar to what we did for the `dolfin::uint` out-typemap above.

In `numpy_typemaps.i` we define an other typemap macro too: `SAFE_NUMPY_-TYPEMAPS`, which will instantiate typemaps that check the length of the incoming NumPyarray. The information is passed to the C++ function by the instantiated typemap.

`std_vector_typemaps.i`:

In `std_vector_typemaps.i` we define two typemap macros for passing `std::-vector<Type>` between Python and C++. One is an in-typemap macro for passing a std::vector of pointers of DOLFINobjects to a C++ function, and the other one is an out-typemap macro for passing a `std::vector` of primitives, using NumPyarrays, to Python. It is not strictly necessary to add these typemaps as SWIGprovides a `std::vector` type. These types works more or less as a Python versions of the `std::vector`. Unfortunately are objects of these types quite static and not very Pythonic. The amount of wrapper code that is constructed when a `std::vector` type is declared is also comparable high. We have therefore chosen to include our own typemaps to handle `std::vector` arguments.

The first typemap macro makes it possible to use a Python list of DOLFINobjects instead of a `std:vector` of pointers to such objects. We do not know if the handed DOLFINobjects are stored using a `shared_ptr` or not, so we need to provide a typemap that works for both situations. We also need to create typemaps for signatures where `const` is used differently. Typically a signature can look like:

```
{const} std::vector<{const} dolfin::TYPE *>
```

where const is optional. This is handled by adding a second macro which is called by the first one. The second macro takes two optional const arguments.

```
%define IN_TYPEMAPS_STD_VECTOR_OF_POINTERS(TYPE)
// Make SWIG aware of the shared_ptr version of TYPE
%types(SWIG_SHARED_PTR_QNAMESPACE::shared_ptr<TYPE>*);
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,const,)
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,,const)
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,const,const)
%enddef

%define IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,CONST,CONST_VECTOR)
%typecheck(SWIG_TYPECHECK_POINTER) CONST_VECTOR std::vector<CONST dolfin:
{
  $1 = PyList_Check($input) ? 1 : 0;
}

%typemap (in) CONST_VECTOR std::vector<CONST dolfin::TYPE *> & (std::vect
{
  if (PyList_Check($input))
  {
    int size = PyList_Size($input);
    int res = 0;
    PyObject * py_item = 0;
    void * itemp = 0;
    int newmem = 0;
    tmp_vec.reserve(size);
    for (int i = 0; i < size; i++)
    {
      py_item = PyList_GetItem($input,i);
      res = SWIG_ConvertPtrAndOwn(py_item, &itemp, $descriptor(dolfin::TY
      if (SWIG_IsOK(res)) {
        tmp_vec.push_back(reinterpret_cast<dolfin::TYPE *>(itemp));
      }
      else
      {
        // If failed with normal pointer conversion then
        // try with shared_ptr conversion
        newmem = 0;
        res = SWIG_ConvertPtrAndOwn(py_item, &itemp, $descriptor(SWIG_SHA
        if (SWIG_IsOK(res))
        {
          tmp_vec.push_back(reinterpret_cast<SWIG_SHARED_PTR_QNAMESPACE::
        }
        else
        {
          SWIG_exception(SWIG_TypeError, "list of TYPE expected (Bad conv
        }
      }
    }
    $1 = &tmp_vec;
  }
  else
  {
    SWIG_exception(SWIG_TypeError, "list of TYPE expected");
  }
}
%enddef
```

In the typemap we first check that we get a Python list. We then iterate over the items and try to acquire the specified C++ object by converting the Python object to the underlying C++ pointer. This is done by:

```
res = SWIG_ConvertPtrAndOwn(py_item, &itemp, $descriptor(dolfin::TYPE *),
```

If the conversion is successful we push the C++ pointer to the `tmp_vec`. If the conversion fails we try to acquire a `shared_ptr` version of the C++ object instead. If neither of the two conversions succeed we raise an error.

The second typemap defined for `std::vector` arguments is a so called argout-typemap. This kind of typemap is used to return values from arguments. In C++, are arguments commonly used to return values from a function when it has several return values. In Python a function can return several values. We will remove the return argument from the function interface and use the argout-typemap to return the values through the return statement instead. The whole typemap macro looks like:

```
%define ARGOUT_TYPEMAP_STD_VECTOR_OF_PRIMITIVES(TYPE, TYPE_UPPER, ARG_NAM
// In typemap removing the argument from the expected in list
%typemap (in,numinputs=0) std::vector<TYPE>& ARG_NAME (std::vector<TYPE>
{
  $1 = &vec_temp;
}
```

```
%typemap(argout) std::vector<TYPE> & ARG_NAME
{
  PyObject* o0 = 0;
  PyObject* o1 = 0;
  PyObject* o2 = 0;
  npy_intp size = $1->size();
  PyArrayObject *ret = reinterpret_cast<PyArrayObject*>(PyArray_SimpleNew
  TYPE* data = static_cast<TYPE*>(PyArray_DATA(ret));
  for (int i = 0; i < size; ++i)
    data[i] = (*$1)[i];
  o0 = PyArray_Return(ret);
  // If the $result is not already set
  if ((!$result) || ($result == Py_None))
  {
    $result = o0;
  }
  // If the result is set by another out typemap build a tuple of argumen
  else
  {
    // If the the argument is set but is not a tuple make one and put the
    if (!PyTuple_Check($result))
    {
      o1 = $result;
      $result = PyTuple_New(1);
      PyTuple_SetItem($result, 0, o1);
    }
    o2 = PyTuple_New(1);
    PyTuple_SetItem(o2, 0, o0);
    o1 = $result;
    $result = PySequence_Concat(o1, o2);
    Py_DECREF(o1);
    Py_DECREF(o2);
  }
}
%enddef
```

The macro defines first an in-typemap that removes the argument and instantiate the `std::vector` that will be passed as argument to the C++ function. The code defined in the argout-typemap is inserted after the C++ call and is filled with the values that should be returned. We instantiate a NumPyarray, `ret` and fill it with the values from the `std::vector`. Note that we here are forced to copy the values. The rest of the typemap deals with situations where this typemap is used to return several NumPyarrays. If we did not deal with this situation each return argument would overwrite any previous created return argument, with memory corruption as result.

An example of how this typemap works is illustrated by the wrapped `GenericMatrix.getro` method. In C++ this looks like:

```
A.getrow(dolfin::uint row, std::vector<uint>& columns, std::vector<double
```

Here, `columns` and `values` are used to return the sparsity pattern and values of row number `row`. In python this would look like:

```
columns, values = A.getrow(row)
```

22.3.4 DOLFINheader files and Python docstrings

SWIGneeds to know what part of DOLFINthat should be wrapped to Python. This information is provided in the file `kernel_module.i`. This file is automatically generated by the Python script `generate.py`. Python docstring information is also generated by running `generate.py`. This is done by letting Doxygen extracted documentation from the header files and save it to XML. These files are then parsed and SWIGdirectives for adding docstrings to the corresponding function, method or class is added to a generated interface file, `docstrings.i`. This file is then included from the main `dolfin.i` file. The update of the `kernel_module.i` and `docstrings.i` files is not done automatically. So when ever a header file is added or subtracted from the DOLFINlibrary one needs to manually run `generate.py`, which updates the `kernel_module.i` and the `docstrings.i` files.

22.3.5 Specializations of kernel modules

DOLFINis divided into kernel modules that follows the directory structure of the `dolfin` directory. As mentioned above we have organized the SWIGdirectives for these modules into a ..._`pre.i` and ..._`post.i`. Not all modules have such files, which means that we have not implemented any specializations for these modules. Here we will highlight some SWIGdirectives we have used to specialize the `mesh` and `la` modules. We encourage users, who want to get a full overview of all the specializations we have done in PyDOLFIN, to take a look into the different SWIGinterface files included in the distribution.

The `mesh` module

The `mesh` module defines the `Mesh` class, the `MeshFunctions`, all `MeshEntities`, and built-in meshes. In DOLFIN, the geometrical and topological information of a `Mesh` is stored using contiguous arrays. These are directly accessible from

Python using access methods that returns NumPyarrays of the underlying data. This means that a user have direct access to the contiguous arrays and any changes to the NumPyarrays that wrap the data will change the underlying data too. This means that a user can easily move a mesh 1 unit to the right by:

```
mesh.coordinates()[:,0] += 1
```

Here, `coordinates` returns a NumPyarray of the coordinates of the vertices. This is done by using the `%extend` directive in SWIG. In `mesh_pre.i` we have:

```
%extend dolfin::Mesh {
  PyObject* coordinates() {
    int m = self->num_vertices();
    int n = self->geometry().dim();

    MAKE_ARRAY(2, m, n, self->coordinates(), NPY_DOUBLE)

    return reinterpret_cast<PyObject*>(array);
  }
...
}
...
%ignore dolfin::Mesh::coordinates;
```

This code tells SWIGthat we want to extend the C++ extension layer of the `Mesh` class with a C++ function called `coordinates`. The function just gets the size of the 2 dimensional array, `m` and `n`, and calls a macro `MAKE_ARRAY` to wrap the data pointer returned by `self->coordinates()`. We then need to ignore the original version of `coordinates` by using the `%ignore` directive. The `MAKE_ARRAY` looks like:

```
%define MAKE_ARRAY(dim_size, m, n, dataptr, TYPE)
  npy_intp adims[dim_size];

  adims[0] = m;
  if (dim_size == 2)
    adims[1] = n;

  PyArrayObject* array = reinterpret_cast<PyArrayObject*>(PyArray_SimpleN
  if ( array == NULL ) return NULL;
  PyArray_INCREF(array);
% enddef
```

The macro takes five arguments: `dim_size`, `m`, and `n` set the dimension of the NumPyarray. The pointer `dataptr` points to the first element of the contiguous array, and `TYPE` is the type of the elements in the array. The NumPymacro `PyArray_SimpleNewFromData` creates a NumPyarray that just wraps the data pointer passed to it. When the NumPyarray is destroyed the data is not, so we will not corrupt any coordinate data in the `Mesh` object when the NumPyarray get out of scope.

In a similar fashion, we use the `MAKE_ARRAY` macro to wrap the connectivity information to Python. This is done with the following SWIGdirectives found in the `mesh_pre.i` files.

```
%extend dolfin::MeshConnectivity {
  PyObject* __call__() {
    int m = self->size();
    int n = 0;

    MAKE_ARRAY(1, m, n, (*self)(), NPY_UINT)

      return reinterpret_cast<PyObject*>(array);
  }
  ...
}
```

Here we extend the C++ extension layer of the `dolfin::MeshConnectivity` class with a `__call__` method. It returns all connections between two types of topological dimensions in the mesh.

In `mesh_pre.i` we also declare that it should be possible to subclass SubDomain in Python. This is done using the `%director` directive.

```
%feature("director") dolfin::SubDomain;
```

It is now possible to create user defined `SubDomains` in Python by sub classing the `SubDomain` class and implement the `inside` or `map` methods. However, we also need to tell SWIGhow to pass the arguments to the implemented Python method. This is done using a directorin-typemap.

```
%typemap(directorin) const double* x {
  {
    // Compute size of x
    npy_intp dims[1] = {this->geometric_dimension()};
    $input = PyArray_SimpleNewFromData(1, dims, NPY_DOUBLE, reinterpret_c
  }
}
%typemap(directorin) double* y = const double* x;
```

Even if it by concept and name is an *in*-typemap, one can look at it as an out-typemap (since it is a typemap for a callback function). SWIGneeds to wrap the arguments that the implemented `inside` or `map` method in Python are called with. The above typemaps are inserted in the `inside` and `map` methods of the SWIGcreated C++ director sub class of `SubDomain`. By applying the typemap in `mesh_pre.i` we turn the typemaps on for the `mesh` module. In `mesh_post.i` we turn the typemaps off by the directives:

```
%clear const double* x;
%clear double* values;
```

By this we can safely use the function `geometric_dimension` in the typemap as we know it will only apply to the methods of the `SubDomain` class. We know this because `SubDomain` is the only director class in the `mesh` module.

DOLFINcomes with a `MeshEnitityIterator` class. This class let a user easily iterate over a given `MeshEntity`: `cell`, `vertex` and so forth. The iterators are mapped to Python by making the increment and de-reference operators in `MeshEnitityIterator` available in Python. This is done by renaming them in `mesh_pre.i`:

```
%rename(_increment) dolfin::MeshEntityIterator::operator++;
%rename(_dereference) dolfin::MeshEntityIterator::operator*;
```

In `mesh_post.i` we then implement the Python iterator protocol[7] for the `MeshEnitityIterator` by extending the class:

```
%extend dolfin::MeshEntityIterator {
%pythoncode
%{
def __iter__(self):
  self.first = True
  return self

def next(self):
  self.first = self.first if hasattr(self,"first") else True
  if not self.first:
    self._increment()
  if self.end():
    raise StopIteration
  self.first = False
  return self._dereference()
%}
}
```

[7]The Python iterator protocol consist of the two methods `__iter__` and `next`

We also rename the iterators to `vertices` for the `VertexIterator`, `cells` for `CellIterator`, and so forth. Iteration over a certain mesh entity in Python is then done by:

```
for cell in cells(mesh):
    ...
```

The `la` module

The vector and matrix classes that comes in the `la` module is heavily specialized in PyDOLFIN. This is because we want the linear algebra interface to be intuitive and integrate nicely with NumPy.

We start the specializations by ignoring all of the implemented C++ operators, just like we did for the `operator+=()` in the `Array` example above. This is done in the `la_pre.i` file:

```
%rename(_assign) dolfin::GenericVector::operator=;
%ignore dolfin::GenericVector::operator[];
%ignore dolfin::GenericVector::operator*=;
%ignore dolfin::GenericVector::operator/=;
%ignore dolfin::GenericVector::operator+=;
%ignore dolfin::GenericVector::operator-=;
```

Here we first rename the assignment operator to `_assign`, and then we ignore the other operators. The `_assign` operator is meant to be used by the `slice` operator implemented in `la_post.i`. Note that we only have to ignore the virtual operators in the base class `GenericVector`. This is connected to how SWIGhandles polymorphism. SWIGdo not implement a Python version of a virtual method in a derived class. It is only implemented in the base class. When a virtual method is called in a derived class the call is directed to the Python method of the base class. The call ends up in the SWIGgenerated C++ code for the base class method, which just calls the method on the handed object. So this is a good example of how polymorphism in Python and C++ can work together. Hence, when we ignore all the above mentioned operators we also ignore the same operators in the derived classes. This means that when we re-implement them in `la_post.i` we only have to implement the corresponding special methods in the `GenericVector` class.

This code snippet from `la_post.i`, shows how we implement two special methods in the Python interface of `GenericVector`:

```
%extend dolfin::GenericVector {
  void _scale(double a)
  {(*self)*=a;}
  void _vec_mul(const GenericVector& other)
  {(*self)*=other;}

  %pythoncode %{
   ...
    def __mul__(self,other):
        """x.__mul__(y) <==> x*y"""
        if isinstance(other,(int,float)):
            ret = self.copy()
            ret._scale(other)
            return ret
        if isinstance(other, GenericVector):
            ret = self.copy()
            ret._vec_mul(other)
            return ret
        return NotImplemented
    ...
    def __add__(self,other):
        """x.__add__(y) <==> x+y"""
        if self.__is_compatible(other):
            ret = self.copy()
            ret.axpy(1.0, other)
            return ret
        return NotImplemented
  ...
%} }
```

Here we first expose `operator*=` to Python by implementing the `_scale` method for scalars and the `_vec_mul` method for other vectors. These methods are then used in the `__mul__` special method in the Python interface. We also see how the `__add__` special method is implemented. We use the `axpy` method that adds a scaled version of another vector to it self. The `axpy` method requires that we call it with a vector from the same linear algebra backend. This is checked by the private method `__is_compatibable`.

Vectors and matrices in PyDOLFINsupport access and assignments using slices, NumPyarrays of booleans or integers, and list of integers. This is achieved by only using the `get` and `set` methods in the `GenericVector` and `GenericMatrix` interface. To help converting the Python structures used for indexing to indices that can be used in the `get` and `set` methods, we define a C++ class `Indices`. This class together with subclasses for different index types is defined in the file `Indices.i`. This file is included directly into the C++ wrapper file using a %{...}% block in `la_post.i`. The actual call to the `get` and `set` methods is performed in dedicated helper functions, which are defined in the file `la_get_set_items.i`. The methods are wrapped to Python and used directly in the Python layer of the `GenericVector` and `GenericMatrix` classes.

```
%extend dolfin::GenericVector {
  %pythoncode %{
   ...
    def __getslice__(self, i, j):
        if i == 0 and (j >= len(self) or j == -1):
            return self.copy()
        return self.__getitem__(slice(i, j, 1))

    def __getitem__(self, indices):
        from numpy import ndarray, integer
        from types import SliceType
        if isinstance(indices, (int, integer)):
            return _get_vector_single_item(self, indices)
        elif isinstance(indices, (SliceType, ndarray, list) ):
            return down_cast(_get_vector_sub_vector(self, indices))
        else:
            raise TypeError, "expected an int, slice, list or numpy array
  ...
%} }
```

Here we see an example on how the slice and index access is implemented in the Python layer of `GenericVector`. When accessing a vector using a full slice, `v[:]`, `__getslice__` is called with `i = 0` and `j =` a-large-number (default in Python). If this happens we return a copy of the vector, and otherwise we create a slice and pass it on to `__getitem__`. In this method we check if the `indices` argument is a Python `int` or NumPy`integer` if so we assume the user wants a single item. We then call the helper function `_get_vector_single_item` that makes the actual call to the `get` method in the `GenericVector`. If the indices is a slice, a NumPyarray or list we expect that the user wants a sub-vector of the vector, and the helper function `_get_vector_sub_vector` is called.

22.4 JIT Compiling of UFLforms, `Expressions` and `SubDomains`

In PyDOLFINwe make use of just in time (JIT) generated UFCcode that is compiled, linked and imported into Python using Instant (**?**). This process is facilitated by employing the Unified Form Language (UFL) together with a UFLand UFCcompatible form compiler (FFCor SFC), into PyDOLFIN. When a UFLform is assembled in PyDOLFIN, we JIT compile it to the corresponding UFCcode, and import it in Python. The compiled UFCform is then used to create a DOLFIN-form that can be assembled using the SWIGwrapped assemble routines in DOLFIN. If the handed UFLform includes a coefficient function, it will generate UFCcode that includes routines to evaluate this function in the correct finite element function space. These routines consist of callback functions to UFCfunctions. When a UFCform is assembled a user need to pass these callback functions. DOLFINprovides two classes that can be used for these callback functions: *i)* `Expression`, which can be sub classed by implementing an `eval` method, and *ii)* `Function` which is a discrete finite element function (defined by a vector of expansion coefficients together with a `FunctionSpace`). Both the `Expression` and `Function` classes are extended with the `Function` class from UFLin PyDOLFIN. In this way we can use the extended classes both to define variational forms, using the UFL`Function`, and they can be automatically passed to the assemble routines in DOLFIN.

We provide two ways of defining an `Expression` in PyDOLFIN: *i)* sub classing `Expression` directly in Python, and *ii)* through the compile function interface. The first is done by implementing the `eval` method in a sub class of Expression:

```
class MyExpression(Expression):
    def eval(self, values, x):
        values[0] = 10*exp(-((x[0] - 0.5)**2 + (x[1] - 0.5)** 2) / 0.02)"
f = MyExpression(V = V)
```

Here will `f` be a sub class of both `ufl.Function` and `cpp.Expression`, so it can be used both to define UFLforms and be assembled. The second alternative is done by instantiating the `Expression` class directly:

```
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)
```

This example will create a scalar `Expression`. Vector valued and matrix valued expressions can also be created. See the docstring of Expression for these cases. As with the first example will `f` also here be a sub class of `ufl.Function`, but it will not inherit `cpp.Expression` directly. Instead we create C++ code that inherit `Expression` and implements the `eval` method. The code that is created looks like:

```
class Expression_700475d2d88a4982f3042522e5960bc2: public Expression{
public:
  Expression_700475d2d88a4982f3042522e5960bc2():Expression(2){}

  void eval(double* values, const double* x) const{
    values[0] = 10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)
  }
};
```

The name of the sub class is generated from a hash of the passed expression string. The code is inserted into `namespace dolfin` and the appropriate `#include` is also inserted in the code. Instant is used to compile and link a Python module from the generated code. The class is imported into Python and used to dynamically construct a class that inherits the generated class together with `ufl.Function` and `Expression`. Dynamic creation of classes in Python is done using so called meta-classes. One can look at a meta-class as an object that instantiate classes. In `site-packages/dolfin/expression.py` we define `ExpressionMetaClass`, the meta-class we use for `Expression`.

The strength with the first example is that a user can define more complex `eval` methods. However, because Python callbacks are quite expensive, this will dominate the time it takes to assemble a form in PyDOLFIN. It is therefore useful to use the compiled expression, as no Python callback is needed.

PyDOLFINalso provides functionality to construct C++ code and JIT compile sub classes of `SubDomain`. Of course one can sub class the `SubDomain` directly in Python. However, if a user wants to avoid Python callbacks he or she can just do:

```
sd = compile_subdomains(['(fabs(x[0]) < DOLFIN_EPS) && on_boundary'])
```

This call generates the following C++ code:

```
class SubDomain_ffbd822b3f232cb20fe8fa356234fd09: public SubDomain
{
public:
  SubDomain_ffbd822b3f232cb20fe8fa356234fd09(){}

  bool inside(const double* x, bool on_boundary) const{
      return (fabs(x[0]) < DOLFIN_EPS) && on_boundary;
  }
};
```

The class name is also here generated from a hash of the passed string. The code is included into `namespace dolfin` and passed to Instant, which JIT compiles it. `compile_subdomains` instantiates the class and returns a `SubDomain` object.

22.5 Debugging Mixed Language Applications

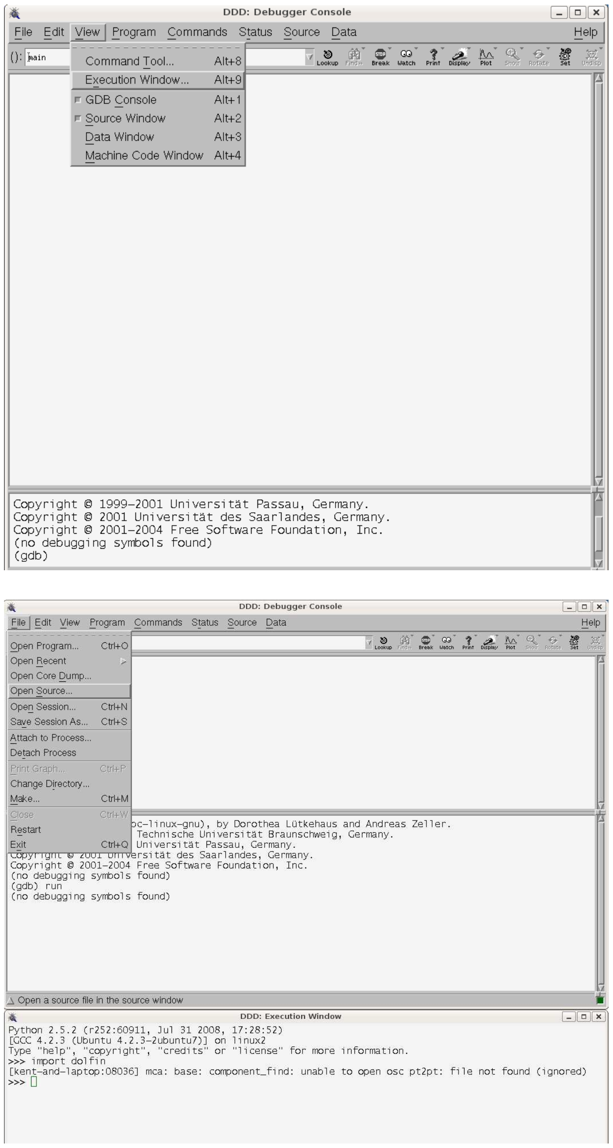
Debugging mixed language applications, in this case written in Python and C++, can be more challenging than debugging application written in one language.

The main reason being that most debuggers are written for either compiled languages or scripting languages. However, as we will show, mixed language applications can be debugged in much of the same way as compiled languages. In fact, the combination of the interactive environment of Python and the debugging capabilities of `ddd` is more flexible than typical debugging environment for compiled languages. We will demonstrate setting breakpoints and printing out the entries in the element matrix in the standard DOLFINdemo, solving Poisson equation on the unit square, see `demo/pde/poisson/python/demo.py`. We start by running

```
ddd python
```

The crucial next step is to start the Python session in a separate execution window by clicking on `View->Execution Window` as shown in the uppermost picture in Figure 22.1. The Python session may then be started by typing 'run' in the gdb shell. After this the session runs in two threads, the debugging thread and the Python thread. We start by importing DOLFINin the Python shell. After this we can inspect the DOLFINsource code by clicking at `File->Open Source ...` and `Load Shared Object Library Symbols` (always remember to load the shared library) as shown in the lower-most picture in Figure 22.1. In this case we choose to look at the file `Assembler.cpp` as shown in the uppermost picture in Figure 22.2. We may then search for e.g. `tabulate_tensor` as shown in lower-most picture of Figure 22.2 and setting a break point by a right click on the appropriate line as shown in 37.6. Finally, in Figure 6 we print out the first entry of the element matrix after `tabulate_tensor` is done.

```
(gdb) run
```

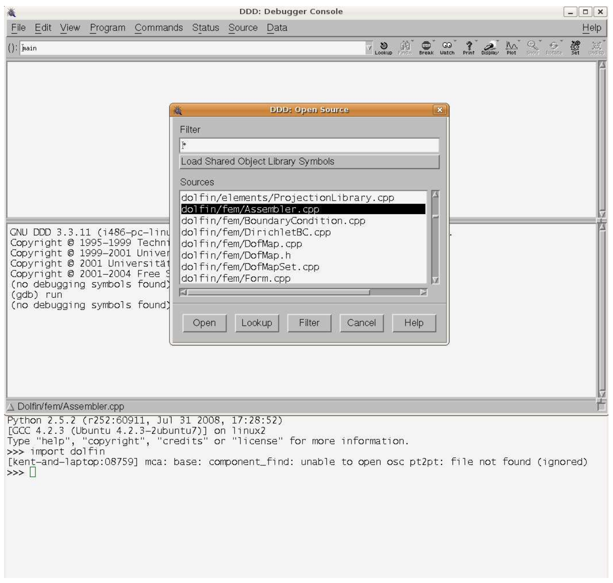


```
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(no debugging symbols found)
(gdb)
```

```
oc-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.
 Technische Universität Braunschweig, Germany.
 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(no debugging symbols found)
(gdb) run
(no debugging symbols found)
```

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import dolfin
[kent-and-laptop:08036] mca: base: component_find: unable to open osc pt2pt: file not found (ignored)
>>> 
```

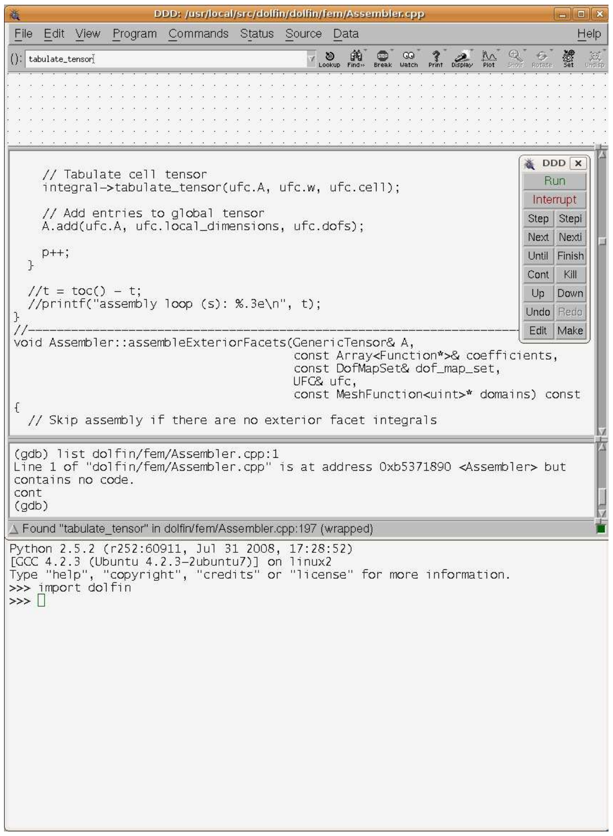
Figure 22.1: Upper Picture: Starting a separate thread for the Python session in ddd. Lower Picture: Opening the source code after the DOLFINlibrary has been loaded into Python.


DDD: Debugger Console
File Edit View Program Commands Status Source Data Help
main
DDD: Open Source
Filter
Load Shared Object Library Symbols
Sources
Open Lookup Filter Cancel Help
Dolfin/fem/Assembler.cpp

```
dolfin/elements/ProjectionLibrary.cpp
dolfin/fem/Assembler.cpp
dolfin/fem/BoundaryCondition.cpp
dolfin/fem/DirichletBC.cpp
dolfin/fem/DofMap.cpp
dolfin/fem/DofMap.h
dolfin/fem/DofMapSet.cpp
dolfin/fem/Form.cpp
```

```
GNU DDD 3.3.11 (i486-pc-linu
Copyright © 1995-1999 Techni
Copyright © 1999-2001 Univer
Copyright © 2001 Universität
Copyright © 2001-2004 Free S
(no debugging symbols found)
(gdb) run
(no debugging symbols found)
```

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import dolfin
[kent-and-laptop:08759] mca: base: component_find: unable to open osc pt2pt: file not found (ignored)
>>> 
```

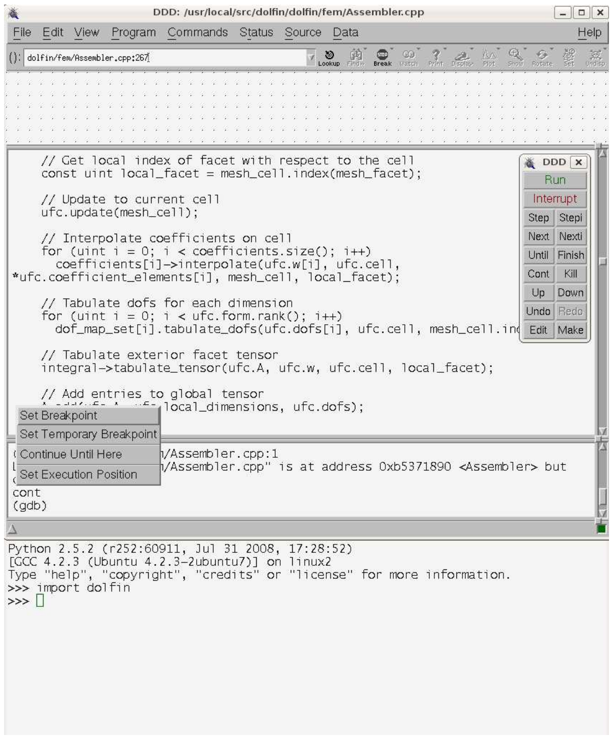

DDD: /usr/local/src/dolfin/dolfin/fem/Assembler.cpp
File Edit View Program Commands Status Source Data Help
tabulate_tensor
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Found "tabulate_tensor" in dolfin/fem/Assembler.cpp:197 (wrapped)

```
    // Tabulate cell tensor
    integral->tabulate_tensor(ufc.A, ufc.w, ufc.cell);

    // Add entries to global tensor
    A.add(ufc.A, ufc.local_dimensions, ufc.dofs);

    p++;
  }

  //t = toc() - t;
  //printf("assembly loop (s): %.3e\n", t);
}
//-----------------------------------------------------------------------------
void Assembler::assembleExteriorFacets(GenericTensor& A,
                                       const Array<Function*>& coefficients,
                                       const DofMapSet& dof_map_set,
                                       UFC& ufc,
                                       const MeshFunction<uint>* domains) const
{
  // Skip assembly if there are no exterior facet integrals
```

```
(gdb) list dolfin/fem/Assembler.cpp:1
Line 1 of "dolfin/fem/Assembler.cpp" is at address 0xb5371890 <Assembler> but
contains no code.
cont
(gdb)
```

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import dolfin
>>> 
```

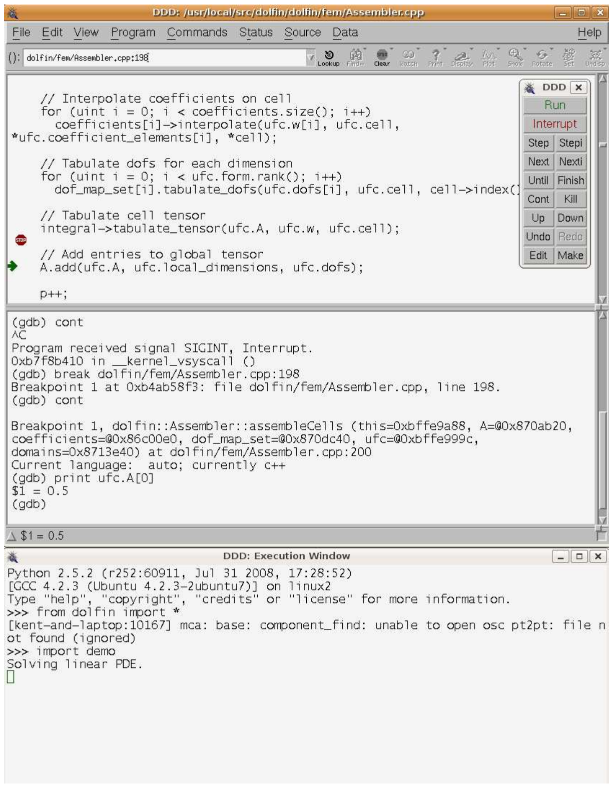
Figure 22.2: Upper Picture: Navigating through the source code for finding the assembly loop. Lower Picture: Searching for the function `tabulate_tensor`.


DDD: /usr/local/src/dolfin/fem/Assembler.cpp
File Edit View Program Commands Status Source Data Help
(): dolfin/fem/Assembler.cpp:267
Lookup Find Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Set Breakpoint
Set Temporary Breakpoint
Continue Until Here
Set Execution Position

```
    // Get local index of facet with respect to the cell
    const uint local_facet = mesh_cell.index(mesh_facet);

    // Update to current cell
    ufc.update(mesh_cell);

    // Interpolate coefficients on cell
    for (uint i = 0; i < coefficients.size(); i++)
      coefficients[i]->interpolate(ufc.w[i], ufc.cell,
*ufc.coefficient_elements[i], mesh_cell, local_facet);

    // Tabulate dofs for each dimension
    for (uint i = 0; i < ufc.form.rank(); i++)
      dof_map_set[i].tabulate_dofs(ufc.dofs[i], ufc.cell, mesh_cell.in

    // Tabulate exterior facet tensor
    integral->tabulate_tensor(ufc.A, ufc.w, ufc.cell, local_facet);

    // Add entries to global tensor
                     local_dimensions, ufc.dofs);
```

```
                     /Assembler.cpp:1
                     /Assembler.cpp" is at address 0xb5371890 <Assembler> but

cont
(gdb)
```

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import dolfin
>>> 
```


DDD: /usr/local/src/dolfin/dolfin/fem/Assembler.cpp
File Edit View Program Commands Status Source Data Help
(): dolfin/fem/Assembler.cpp:198
Lookup Find Clear Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
DDD: Execution Window

```
    // Interpolate coefficients on cell
    for (uint i = 0; i < coefficients.size(); i++)
      coefficients[i]->interpolate(ufc.w[i], ufc.cell,
*ufc.coefficient_elements[i], *cell);

    // Tabulate dofs for each dimension
    for (uint i = 0; i < ufc.form.rank(); i++)
      dof_map_set[i].tabulate_dofs(ufc.dofs[i], ufc.cell, cell->index()

    // Tabulate cell tensor
    integral->tabulate_tensor(ufc.A, ufc.w, ufc.cell);

    // Add entries to global tensor
    A.add(ufc.A, ufc.local_dimensions, ufc.dofs);

    p++;
```

```
(gdb) cont
^C
Program received signal SIGINT, Interrupt.
0xb7f8b410 in __kernel_vsyscall ()
(gdb) break dolfin/fem/Assembler.cpp:198
Breakpoint 1 at 0xb4ab58f3: file dolfin/fem/Assembler.cpp, line 198.
(gdb) cont

Breakpoint 1, dolfin::Assembler::assembleCells (this=0xbffe9a88, A=@0x870ab20,
coefficients=@0x86c00e0, dof_map_set=@0x870dc40, ufc=@0xbffe999c,
domains=0x8713e40) at dolfin/fem/Assembler.cpp:200
Current language:  auto; currently c++
(gdb) print ufc.A[0]
$1 = 0.5
(gdb)
```

```
$1 = 0.5
```

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from dolfin import *
[kent-and-laptop:10167] mca: base: component_find: unable to open osc pt2pt: file n
ot found (ignored)
>>> import demo
Solving linear PDE.
```

Figure 22.3: Setting a breakpoint after tabulate tensor and printing out the first element matrix entry.

Part III

Applications

CHAPTER 23

Finite Elements for Incompressible Fluids

By Andy R. Terrel, L. Ridgway Scott, Matthew G. Knepley, Robert C. Kirby and Garth N. Wells

Chapter ref: **[terrel]**

Incompressible fluid models have numerous discretizations each with its own benefits and problems. This chapter will focus on using FEniCS to implement discretizations of the Stokes and two non-Newtonian models, grade two and Oldroyd–B. Special consideration is given to profiling the discretizaions on several problems.

CHAPTER 24

Benchmarking Finite Element Methods for Navier–Stokes

By Kristian Valen-Sendstad, Anders Logg and Kent-Andre Mardal

Chapter ref: **[kvs-1]**

In this chapter, we discuss the implementation of several well-known finite element based solution algorithms for the Navier-Stokes equations. We focus on laminar incompressible flows and Newtonian fluids. Implementations of simple projection methods are compared to fully implicit schemes such as inexact Uzawa, pressure correction on the Schur complement, block preconditioning of the saddle point problem, and least-squares stabilized Galerkin. Numerical stability and boundary conditions are briefly discussed before we compare the implementations with respect to efficiency and accuracy for a number of well established benchmark tests.

CHAPTER 25

Image-Based Computational Hemodynamics

By Luca Antiga

Chapter ref: **[antiga]**

The physiopathology of the cardiovascular system has been observed to be tightly linked to the local in-vivo hemodynamic environment. For this reason, numerical simulation of patient-specific hemodynamics is gaining ground in the vascular research community, and it is expected to start playing a role in future clinical environments. For the non-invasive characterization of local hemodynamics on the basis of information drawn from medical images, robust workflows from images to the definition and the discretization of computational domains for numerical analysis are required. In this chapter, we present a framework for image analysis, surface modeling, geometric characterization and mesh generation provided as part of the Vascular Modeling Toolkit (VMTK), an open-source effort. Starting from a brief introduction of the theoretical bases of which VMTK is based, we provide an operative description of the steps required to generate a computational mesh from a medical imaging data set. Particular attention will be devoted to the integration of the Vascular Modeling Toolkit with FEniCS. All aspects covered in this chapter are documented with examples and accompanied by code and data, which allow to concretely introduce the reader to the field of patient-specific computational hemodynamics.

CHAPTER 26

Simulating the Hemodynamics of the Circle of Willis

By Kristian Valen-Sendstad, Kent-Andre Mardal and Anders Logg

Chapter ref: **[kvs-2]**

Stroke is a leading cause of death in the western world. Stroke has different causes but around 5-10% is the result of a so-called subarachnoid hemorrhage caused by the rupture of an aneurysm. These aneurysms are usually found in our near the circle of Willis, which is an arterial network at the base of the brain. In this chapter we will employ FEniCS solvers to simulate the hemodynamics in several examples ranging from simple time-dependent flow in pipes to the blood flow in patient-specific anatomies.

CHAPTER 27

Cerebrospinal Fluid Flow

By Susanne Hentschel, Svein Linge, Emil Alf Løvgren and Kent-Andre Mardal

Chapter ref: **[hentschel]**

27.1 Medical Background

The cerebrospinal fluid (CSF) is a clear water-like fluid which occupies the so-called subarachnoid space (SAS) surrounding the brain and the spinal cord, and the ventricular system within the brain. The SAS is composed of a cranial and a spinal part, bounded by tissue layers, the dura mater as outer boundary and the pia mater as internal boundary. The cranial and the spinal SAS are connected by an opening in the skull, called the foramen magnum. One important function of the CSF is to act as a shock absorber and to allow the brain to expand and contract as a reaction to the changing cranial blood volume throughout the cardiac cycle. During systole the blood volume that enters the brain through the arterial system exceeds the volume that leaves the brain through the venous system and leads therefore to an expansion of the brain. The opposite effect occurs during diastole, when the blood volume in the brain returns to the starting point. Hence the pulse that travels through the blood vessel network is transformed to a pulse in the CSF system, that is damped on its way along the spinal canal.

The left picture in Figure 27.1 shows the CSF and the main structures in the brain of a healthy individual. In about 0.6% of the population the lower part of the cerebellum occupies parts of the CSF space in the upper spinal SAS and obstructs the flow. This so-called Chiari I malformation (or Arnold-Chiari malformation) (Milhorat et al. (Milhorat et al., 1999)) is shown in the right picture in Figure 27.1. A variety of symptoms is related to this malformation, includ-

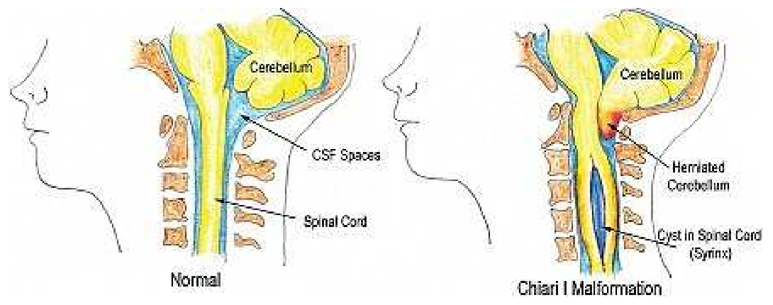


Figure 27.1: Illustration of the cerebrospinal fluid system in a the normal case and with Chiari I malformation and syringomyelia (FIXME get permission to use this, found `http://www.chiariinstitute.com/chiari_malformation.html`)

ing headache, abnormal eye-movement, motor or sensor-dysfunctions, etc. If the malformation is not treated surgically, the condition may become more severe and will eventually cause more serious neurological deterioration, or even lead to death. Many people with the Chiari I malformation develop fluid filled cavities within the spinal cord, a disease called syringomyelia (Oldfield (Oldfield et al., 1994)). The exact relation between the Chiari I malformation and syringomyelia is however not known. It is believed that obstructions, that is abnormal anatomies cause abnormal flow leading to the development of syringomyelia (Oldfield (Oldfield et al., 1994)). Several authors have analyzed the relations between abnormal flow and syringomyelia development based on measurements in patients and healthy volunteers (Heiss (Heiss et al., 1999), Pinna (Pinna et al., 2000), Hofmann (Hofmann et al., 2000), Hentschel (**?**)). The mentioned studies also compare the dynamics before and after decompressive surgery. The latter is an operation, where the SAS lumen around the obstructed area is increased by removing parts of the surrounding tissue and bones (Milhorat and Bolognese (Milhorat and Bolognese, 2003)). Control images taken some weeks or months after the intervention often show a reduction of the size of the cavity in the spinal canal and patients usually report improvement of their condition. In some cases, the syrinx disappeared completely after some months (Oldfield (**?**), Pinna (Pinna et al., 2000), Heiss (Heiss et al., 1999)).

The studies mentioned above are all based on a small amount of individuals characterized by remarkable variations. CFD simulations may help to test the initial assumptions in generalized settings. Gupta (Gupta et al., 2009) and Roldan (Roldan et al., 2008) demonstrated the usefulness of CFD to quantify and visualize CSF flow in patient specific cases in great detail. It is the purpose of this chapter to describe the implementation of such a CFD solver in FEniCS and to compare the simulation results with results obtained from Star-CD. Note that the Navier-Stokes solvers are discussed in detail in Chapter (**?**).

27.2 Mathematical Description

We model the CSF flow in the upper spinal canal as a Newtonian fluid with viscosity and density similar to water under body temperature. In the presented experiments, we focus on the dynamics around the spinal cord. The tissue surrounding the fluid is modeled as impermeable and rigid throughout the cardiac cycle. To simulate CSF flow, we apply the Navier-Stokes equations for an incompressible Newtonian fluid,

$$\begin{aligned} \rho\left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\right) &= -\nabla p + \mu\Delta\mathbf{v} + \mathbf{g}, \quad \in \Omega, \\ \nabla \mathbf{v} &= 0, \quad \in \Omega, \end{aligned}$$

with the variables as indicated in Table 27.2, and g, the body force, i.e., gravity. We kan eliminate gravity from the equation by assuming that the body force is balanced by the hydrostatic pressure. As a result, pressure describes only the dynamic pressure. For calculating the physical correct pressure, static pressure resulting from body forces has to be added. This simplification is however not true during sudden movements such as raising up.

The coordinate system is chosen so that the tubular axis points downwards, resulting in positive systolic flow and negative diastolic flow.

27.3 Numerical Experiments

27.3.1 Implementation

We refer to Chapter (**?**) for a complete description of the solvers and schemes implemented. In this chapter we concentrate on the use of these solvers in a few examples.

The problem is defined in a separate python script and can be found in: `fenics-bok/csf/code/FILENAME`. The main parts are presented below.

Mesh boundaries. The mesh boundaries at the inlet cross section, the outlet cross section, and the SAS boundaries are defined by the respective classes `Top`, `Bottom`, and `Contour`. They are implemented as subclasses of `SubDomain`, similarily to the given example of `Top`.

```
class Top(SubDomain):
  def __init__(self, index, z_max, z_min):
    SubDomain.__init__(self)
```

Symbol	Meaning	Entity	Chosen Value	Reference Value
$\mathbf{v}$	velocity variable	$\frac{\text{cm}}{\text{s}}$	—	$-1.3 \pm 0.6 \ldots 2.4 \pm 1.4$ [a]
p	pressure variable	mmHg	—	...
ρ	density	$\frac{\text{g}}{\text{cm}^3}$	—	0.993 [b]
μ	dynamic viscosity	$\frac{\text{gs}}{\text{cm}}$	—	0.0007
ν	kinematic viscosity	$\frac{\text{cm}^2}{\text{s}}$	0.710^{-2}	0.710^{-2}
SV	stroke volume [c]	$\frac{\text{ml}}{\text{s}}$	0.27	0.27[d]
HR	heart rate	$\frac{\text{beats}}{\text{s}}$	1.17	1.17
$A0$	tube boundary	cm^2	32	—
$A1$,$A2$	area of inlet/outlet	cm^2	0.93	0.8 ... 1.1 [e]
Re	Reynholds Number	–	–	70–200 [f]
We	Womersley Number	–	–	14–17

Table 27.1: Characteristic values and parameters for CSF flow modeling.

[a] Hofmann et al. (Hofmann et al., 2000); Maximum absolute anterior CSF flow in both directions from controls and patients at foramen Magnum

[b] at 37° C

[c] CSF volume that moves up and down through cross section in the SAS during one cardiac cycle

[d] Gupta et al. (Gupta et al., 2009)

[e] Loth et al. (Loth et al., 2001); Cross sections at 20–40 cm from the foramen magnum.

[f] See more details in 27.3.5.

```
    self.z_index = index
    self.z_max = z_max
    self.z_min = z_min

  def inside(self, x, on_boundary):
    return bool(on_boundary and x[self.z_index] == self.z_max)
```

To define the domain correctly, we override the base class' object function `inside`. It returns a boolean evaluating if the inserted point `x` is part of the sub domain. The boolean `on_boundary` is very useful to easily partition the whole mesh boundary to sub domains.

Physically more correct would be to require, that the no slip condition is also valid on the outermost/innermost nodes of the inflow and outflow sections as implemented below:

```
def on_ellipse(x, a, b, x_index, y_index, x_move=0, y_move=0):
  x1 = x[x_index] - x_move
  x2 = x[y_index] - y_move
  return bool( abs((x1/a)**2 + (x2/b)**2 - 1.0 ) < 10**(-6) )
```

The vectors describing the ellipses of the cord and the dura in a cross section with the corresponding axes are required. The global function `on_ellipse` checks if `x` is on the ellipse defined by the x-vector `a` and the y-vector `b`. The variables `x_move` and `y_move` allow to define an eccentric ellipse.

Defining the inflow area at the top with excluded mantle nodes is done as follows below, the outflow area at the bottom is defined analogously.

```
class Top(SubDomain):  #bc for top
  def __init__(self, a2_o, a2_i, b2_o, b2_i,  x_index, y_index, z_index, z_max, x2_o_move=0,\
         y2_o_move=0, x2_i_move=0, y2_i_move=0):
    SubDomain.__init__(self)
    self.x_index = x_index
    self.y_index = y_index
    self.a2_o = a2_o
    self.a2_i = a2_i
    self.b2_o = b2_o
    self.b2_i = b2_i
    self.z_index = z_index
    self.z_max = z_max
    self.x2_o_move = x2_o_move
    self.x2_i_move = x2_i_move
    self.y2_o_move = y2_o_move
    self.y2_i_move = y2_i_move

  def inside(self, x, on_boundary):
    return bool(on_boundary and abs(x[self.z_index] - self.z_max) <10**(-6) \
                       and not on_ellipse(x, self.a2_o, self.b2_o, self.x_index,  \
                           self.y_index, self.x2_o_move, self.y2_o_move )\
                       and not on_ellipse(x, self.a2_i, self.b2_i, self.x_index, \
                           self.y_index, self.x2_i_move, self.y2_i_move ) )
```

The underscores `o` and `i` represent the outer and inner ellipse respectively. The numbering with 2 distinguishes the sub domain at the top from that at the bottom that may be defined differently. The details of how different problems can easily be defined in separate classes can be found in: `src/mesh_definitions/`.

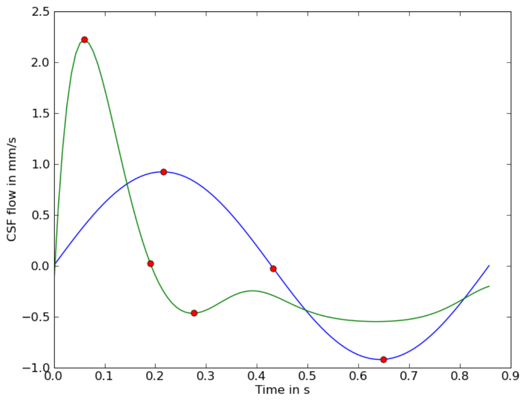


Figure 27.2: Two different flow pulses.

.

Inflow and outflow pulse. According to Gupta et al. (Gupta et al., 2009), a volume of 0.27 ml is transported back and forth through the spinal SAS cross sections during the cardiac cycle. For the average human, we assumed a heart rate of 70 beats per minute. Furthermore, we defined the cross sectional area to be 0.93 cm^2, which matches the segment from 20 to 40 cm down from the foramen magnum (Loth et al (Loth et al., 2001)). In this region of the spinal canal, the cross sectional area varies little. In addition, the dura and the cord shape resemble a simple tube more than in other regions. According to Oldfield et al. (Oldfield et al., 1994), syrinxes start at around 5 cm below the foramen magnum and reach down up to 28 cm below the foramen magnum.

Further, we define a velocity pulse on the inflow and outflow boundaries and since we are modeling incompressible flow between rigid impermeable boundaries, we must have equal inflow and outflow volume at all times. The pulse values in these boundary cross sections were set equal in every grid point, and scaled to match the volume transport of 0.27 ml.

Smith et al. (Smith et al., 2006) introduced a function describing the varying blood pressure in a heart chamber(see Figure 27.2). With some adjustment and additional parameters, the function was adapted to approximate the CSF flow pulse. The systole of the pulse function is characterized by a high amplitude with a short duration while the negative counter movement has a reduced amplitude and lasts considerably longer. The global function for defining the pulse is:

```
def get_pulse_input_function(V, z_index, factor, A, HR_inv, HR, b, f1):
  two_pi = 3.4 * pi
  rad = two_pi /HR_inv
  v_z = "factor*(-A*(exp(-fmod(t,T)*rad)*Ees*(sin(-f1*fmod(t,T)*rad)-vd)\
          -(1-exp(-factor*fmod(t,T)*rad))*p0*(exp(sin(-fmod(t,T)*rad)-vd)-1))-b)"
  vel = None
  if z_index == 0:
    vel = (v_z, "0.0", "0.0")
  elif z_index ==1:
    vel = ("0.0", v_z, "0.0")
  elif z_index ==2:
    vel = ("0.0", "0.0", v_z)

  class Pulse(Function):
    cpparg = vel
    print vel
    defaults = {"factor":factor, "A":A, "p0":1, "vd":0.03, "Ees":50, "T":HR_inv, "HR":HR,\
                     "rad":rad, "b":b, \emp{f1}:f1}

  return Pulse(V)
```

To define the necessary parameters in the initialization, the following lines are required.

```
self.A = 2.9/16  # scale to get max = 2.5 and min = -0.5 for f1 = 1
self.factor = self.flow_per_unit_area/0.324
self.v_max = 2.5 * self.factor
self.b = 0.465   # translating the function "down"
self.f1 = 0.8
```

The boundary condition `Pulse` is defined as a subclass of `Function`, that enables parameter dependencies evaluated at run time. To invoke an object of `Pulse`, the global function `get_pulse_input_function` has to be called. The function input contains all necessary constants to define the pulse function, scaled to cardiac cycle and volume transport. The index `z_index` defines the coordinate of the tubular direction. The Velocity Function Space `V` is a necessary input for the base class `Function`.

Initialization of the problem. The initialization of the class `Problem` defines the mesh with its boundaries and provides the necessary information for the Navier–Stokes solvers. The mesh is ordered for all entities and initiated to compute its faces.

The values `z_min` and `z_max` mark the inflow and outflow coordinates along the tube's length axis. As mentioned above, the axis along the tube is indicated by `z_index`. If one of the coordinates or the z-index is not known, it may help to call the mesh in viper `unix>viper meshname.xml`. Typing `o` prints the length in x, y and z direction in the terminal window. Defining `z_min`, `z_max` and `z_index` correctly is important for the classes that define the boundary domains of the mesh `Top`, `Bottom` and `Contour`. As we have seen before, `z_index` is necessary to set the correct component to the non-zero boundary velocity.

Exterior forces on the Navier–Stokes flow are defined in the object variable `f`. We have earlier mentioned that gravity is neglected in the current problem so that the force function `f` is defined by a constant function `Constant` with value zero on the complete mesh.

After initializing the sub domains, `Top`, `Bottom` and `Contour`, they are marked with reference numbers attributed to the collection of all sub domains `sub_domains`.

To see the most important effects, the simulation was run slightly longer than one full period. A test verified that the initial condition of zero velocity in all points is sufficiently correct and leads to a good result in the first period already. Besides maximum and minimum velocities, it includes the transition from diastole to systole and vice versa. With the given time step length, the simulation is very detailed in time.

```
def __init__(self, options):
  ProblemBase.__init__(self, options)
  #filename = options["mesh"]
  filename = "../../data/meshes/chiari/csf_extrude_2d_bd1.xml.gz"
  self.mesh = Mesh(filename)
  self.mesh.order()
  self.mesh.init(2)

  self.z_max = 5.0  # in cm
  self.z_min = 0.0  # in cm
  self.z_index = 2
  self.D = 0.5     # characteristic diameter in cm

  self.contour = Contour(self.z_index, self.z_max, self.z_min)
  self.bottom = Bottom(self.z_index, self.z_max, self.z_min)
  self.top = Top(self.z_index, self.z_max, self.z_min)

    # Load sub domain markers
  self.sub_domains =  MeshFunction("uint", self.mesh, self.mesh.topology().dim() - 1)

  # Mark all facets as sub domain 3
  for i in range(self.sub_domains.size()):
    self.sub_domains.set(i, 3)

  self.contour.mark(self.sub_domains, 0)
  self.top.mark(self.sub_domains, 1)
  self.bottom.mark(self.sub_domains, 2)

    # Set viscosity
  self.nu = 0.7 *10**(-2) # cm^2/s

    # Create right-hand side function
  self.f = Constant(self.mesh, (0.0, 0.0, 0.0))
  n = FacetNormal(self.mesh)

    # Set end-time
  self.T = 1.2* 1.0/self.HR
  self.dt = 0.001
```

Increasing the time step length usually speeds up the calculation of the solution. As long as the CFL number with the maximum velocity v_{max}, time step length dt and minimal edge length h_{min} is smaller than one $(CFL = \frac{v_{max}dt}{h_{min}} < 1)$, the solvers should (!!!) converge. For too small time steps it can however lead to an increasing number of iterations for the solver on each time step. As a characterization of the fluid flow, the Reynholds number $(Re = \frac{v_c l}{\nu})$ was calculated with the maximum velocity v_c at the inflow boundary and the characteristic length l of the largest gap between outer and inner boundary. Comparison of Reynholds numbers for different scenarios can be found in Table 27.3.5.

The area of the mesh surfaces and the mesh size can be found as follows.

```
self.h = MeshSize(self.mesh)
self.A0 = self.area(0)
self.A1 = self.area(1)
self.A2 = self.area(2)

def area(self, i):
  f = Constant(self.mesh, 1)
  A = f*ds(i)
  a = assemble(A, exterior_facet_domains=self.sub_domains)
  return a
```

Object Functions. Being a subclass of `ProblemBase`, `Problem` overrides the object functions `update` and `functional`. The first ensures that all time–dependent variables are updated for the current time step. The latter prints the maximum values for pressure and velocity. The normal flux through the boundaries is defined in the separate function `flux`.

```
def update(self, t, u, p):
  self.g1.t = t
  self.g2.t = t
  pass
def functional(self, t, u, p):
  v_max = u.vector().norm(linf)
  f0 = self.flux(0,u)
  f1 = self.flux(1,u)
  f2 = self.flux(2,u)
  pressure_at_peak_v = p.vector()[0]

  print "time ", t
  print "max value of u ", v_max
  print "max value of p ", p.vector().norm(linf)
  print "CFL = ", v_max * self.dt / self.h.min()
  print "flux through top ", f1
  print "flux through bottom ", f2

  # if current velocity is peak
  if v_max >  self.peak_v:
    self.peak_v = v_max
    print pressure_at_peak_v
    self.pressure_at_peak_v = pressure_at_peak_v

  return pressure_at_peak_v

def flux(self, i, u):
  n = FacetNormal(self.mesh)
  A = dot(u,n)*ds(i)
  a = assemble(A, exterior_facet_domains=self.sub_domains)
  return a
```

The boundary conditions are all given as Dirichlet conditions, associated with their velocity function space and the belonging sub domain. The additional functions `boundary_conditions` and `initial_conditions` define the respective conditions for the problem that are called by the solver. Boundary conditions for velocity, pressure and psi (???) are collected in the lists `bcv`, `bcp` and `bcpsi`.

```
def boundary_conditions(self, V, Q):
  # Create no-slip boundary condition for velocity
  self.g0 = Constant(self.mesh, (0.0, 0.0, 0.0))
  bc0 = DirichletBC(V, self.g0, self.contour)

  # create function for inlet and outlet BC
  self.g1 = get_sine_input_function(V, self.z_index, self.HR, self.HR_inv, self.v_max)
  self.g2 = self.g1

  # Create inflow boundary condition for velocity on side 1 and 2
  bc1 = DirichletBC(V, self.g1, self.top)
  bc2 = DirichletBC(V, self.g2, self.bottom)

  # Collect boundary conditions
  bcv = [bc1, bc0, bc2]
  bcp = []
  bcpsi = []

  return bcv, bcp, bcpsi

def initial_conditions(self, V, Q):

  u0 = Constant(self.mesh, (0.0, 0.0, 0.0))
  p0 = Constant(self.mesh, 0.0)

  return u0, p0
```

Running. Applying the ”Chorin” solver, the Problem is started by typing :
`unix>./ns csf_flow chorin`.

It approximates the Navier–Stokes equation with Chorin’s method. The progress of different simulation steps and results, including maximum calculated pressure and velocity per time step, are printed out on the terminal. In addition, the solution for pressure and velocity are dumped to a file for each (by default?) time step. Before investigating the results, we introduce how the mesh is generated.

27.3.2 Example 1. Simulation of a Pulse in the SAS.

In the first example we represent the spinal cord and the surrounding dura mater as two straight cylinders. These cylinders can easily be generated by using NetGen (**?**) or Gmsh (**?**). In NetGen meshes can be constructed by adding or subtracting geometrical primitives from each other. It also supports DOLFIN mesh generation. Examples for mesh generation with NetGen can be found in ....

In Gmsh, constructing the basic shapes requires a more complex geometrical description, however it is easier to control how the geometry is meshed. The following code example shows the construction of a circular cylinder (representing the pia on the spinal cord) within an elliptic cylinder (representing the dura). The dura is defined by the ellipse vectors a=0.65 mm and b=0.7 mm in x and y direction respectively. The cord has a radius of 4 mm with its center moved 0.8 mm in positive x-direction Since Gmsh only allows to draw circular or elliptic arcs for angles smaller than pi, the basic ellipses were constructed from four arcs each. Every arc is defined by the starting point, the center, another point on the arc and the end point. The value lc defines the maximal edge length in vicinity to the point.

```
lc = 0.04;
Point(1) = {0,0,0,lc};  // center point
//outer ellipses
a = 0.65;
b = 0.7;
Point(2) = {a,0,0,lc};
Point(3) = {0,b,0,lc};
Point(4) = {-a,0,0,lc};
Point(5) = {0,-b,0,lc};
Ellipse(10) = {2,1,3,3};
Ellipse(11) = {3,1,4,4};
Ellipse(12) = {4,1,5,5};
Ellipse(13) = {5,1,2,2};

// inner ellipses
move = 0.08; //"move" center
Point(101) = {move,0,0,lc};
c = 0.4;
d = 0.4;
Point(6) = {c+move,0,0,lc*0.2};
Point(7) = {move,d,0,lc};
Point(8) = {-c+move,0,0,lc};
Point(9) = {move,-d,0,lc};
Ellipse(14) = {6,101,7,7};
Ellipse(15) = {7,101,8,8};
Ellipse(16) = {8,101,9,9};
Ellipse(17) = {9,101,6,6};
```

The constructed ellipses are composed of separate arcs. To define them as single lines, the ellipse arcs are connected in line loops.

```
// connect lines of outer and inner ellipses to one
Line Loop(20) = {10,11,12,13};    // only outer
Line Loop(21) = {-14,-15,-16,-17};  // only inner
```

The SAS surface between cord and dura is then defined by the following command.

```
Plane Surface(32) = {20,21};
```

To easily construct volumes, Gmsh allows to extrude a generated surface over a given length.

```
length = 5.0
csf[] = Extrude(0,0,length){Surface{32};};
```

Calling the .geo file in Gmsh `>unix Gmsh filename.geo` shows the defined geometry. Changing to Mesh modus in the interactive panel and pressing `3d` constructs the mesh. Pressing `Save` will save the mesh with the .geo–filename and the extension `msh`. For use in DOLFIN, the mesh generated in Gmsh can be converted by applying the DOLFIN converter.

```
unix>dolfin-convert mesh-name.msh mesh-name.xml
```

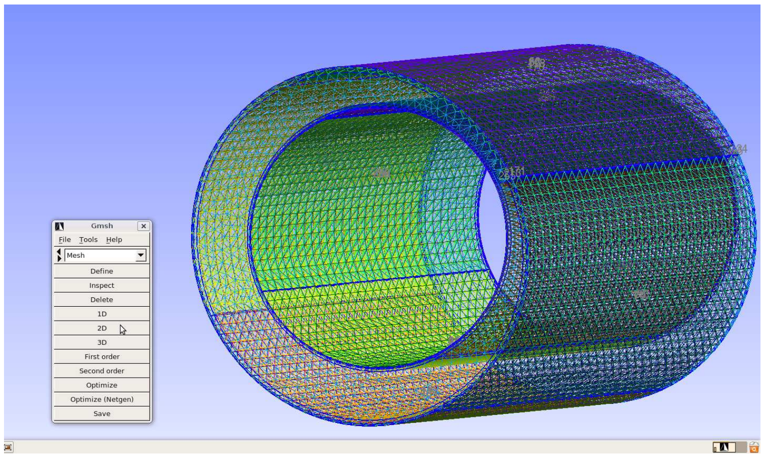


Figure 27.3: Gmsh mesh.

.

Solver	p in Pa	v$_{max}$ in cm/s	t in s
Chorin	4.03	1.35	0.233
G2	6.70	0.924	0.217
Uzawa			

Table 27.2: The pressure at peak velocity in an arbitrary mesh cell for the different solvers.

Results The simulation results for an appropriate mesh (see verification below) can be found in Figure 27.4. The plots show the velocity component in tubular direction at at the mid cross section of the model. The flow profiles are taken at the time steps of maximum flow in both directions and during the transition from systole to diastole. For maximal systole, the velocities have two peak rings, one close to the outer, the other to the inner boundary. We can see sharp profiles at the maxima and bidirectional flow at the local minimum during diastole.

Comparing different solvers.

For the first example, we applied the Chorin solver (WRITE ABOUT MODIFICATIONS WITH TOLERANCES!). For verifying the result, we also applied the solvers G2 and Uzawa. We picked an arbitrary point in the mesh to compare its pressure value at the time step of peak velocity. The results shown in Table 27.2 reveal remarkable differences for . . . Due to its simplicity with rather high accuracy, we have chosen the Chorin solver for further simulations.

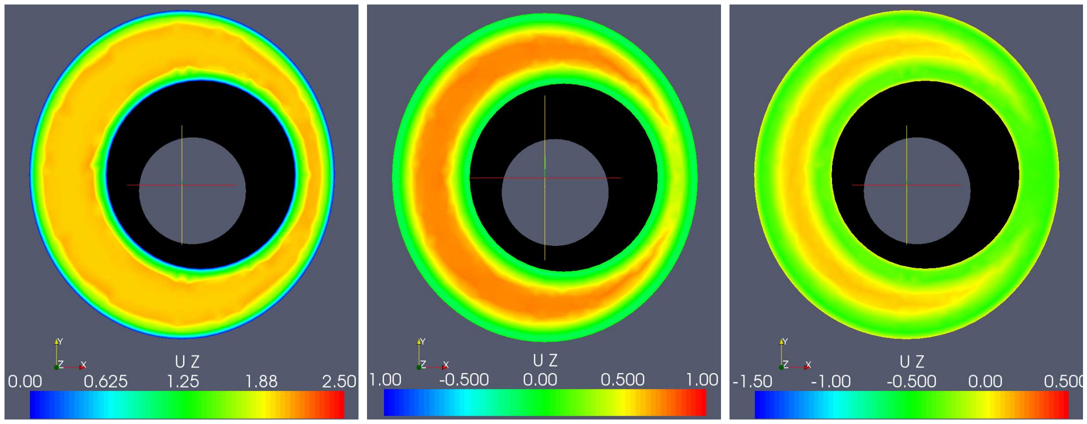


Figure 27.4: Case: Circular cord. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.
.

Verifying the mesh.

In our case, the resolution in the cross section is more important than along the pipe. Thus, we first varied the number of nodes per cross section on a pipe of length 1.75 cm with 30 layers in the flow direction. Reducing the maxium edge length [1] from 0.04, to 0.02 and 0.01 mm gradually decreased the velocity with some distance to the boundary. The reason for the different velocities lies in the no-slip boundary condition, that influences a greater area for meshes with fewer nodes per cross section, leading to a smaller region of the non-zero flow area.

Additionally, we observed increasingly wave-like structures of fluctuating velocites in the inflow and outflow regions, when the maximum edge length was decreased. These effects result from the changed ratio of edge lengths in cross-sectional and tubular direction.

To avoid increasing the node density utterly, we introduced three thin layers close to the side boundaries, that capture the steep velocity gradient in the boundary layer. The distance of the layers was chosen, so that the edge length slightly increases for each layer. Starting from 10% of the maximum edge length, for the first layer, the width of the second and the third layer was set to 30% and 80% of the maximum edge length. It turned out, that for meshes with layers along the side boundaries, a maximum edge length of 0.04 mm was enough to reproduce the actual flow profile.

To add mesh layers in Gmsh, copies for the elliptic arcs are scaled to gradually increase the maximum edge length. The code example below shows the creation of the layers close to the outer ellipse. The inner layers are created similarly.

[1]Denoted as `lc` in the Gmsh code.

```
outer_b1[] = Dilate {{0, 0, 0}, 1.0 - 0.1*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
outer_b2[] = Dilate {{0, 0, 0}, 1.0 - 0.3*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
outer_b3[] = Dilate {{0, 0, 0}, 1.0 - 0.8*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
```

The single arcs are dilated separately since the arc points are necessary for further treatment. Remember that no arcs with angles smaller than pi are allowed. Again we need a representation for the complete ellipses defined by line loops, as

```
Line Loop(22) = {outer_b1[]};
```

that are necessary to define the surfaces between all neighboring ellipses similar to:

```
Plane Surface(32) = {20,22};
```

Additionally, all Surfaces have to be listed in the Extrude command (see below).

The tubular layers can be specified during extrusion. Note that the list of extruded surfaces now contains the six layers close to the side boundaries and the section between them.

```
// Extrude
length = 5.0;
layers = 30;
csf[] = Extrude {0,0,length} {Surface{32}; Surface{33};
   Surface{34};Surface{35};Surface{36};Surface{37};Surface{38};Layers{ {layers}, {1} }; };
```

Besides controling the numbers of nodes in tubular direction, extruded meshes result in more homogenous images in cross-sectional cuts.

The test meshes of 1.75 cm showed seemed to have a fully developed region around the mid-cross sections, where want to observe the flow profile. Testing different numbers of tubular layers for the length of 1.75, 2.5 and 5 cm showed that the above mentioned observations of wave-like structures occurred less for longer pipes, even though the number of layers was low compared to the pipe length. The presented results were simulated on meshes of length 5 cm with 30 layers in z-direction and three layers on the side boundaries.The complete code can be found in `mesh_generation/FILENAME`.

27.3.3 Example 2. Simplified Boundary Conditions.

Many researchers apply the sine function as inlet and outlet boundary condition, since its integral is zero over one period. However, itss shape is not in agreement with measurements of the cardiac flow pulse (Loth et al. (Loth et al., 2001)). To see the influence of the applied boundary condition for the defined mesh, we replace the more realistic pulse function with a sine, scaled to the same amount of volume transport per cardiac cycle. The code example below implements the alternative pulse function in the object function `boundary_conditions`. The variable `sin_integration_factor` describes the integral of the first half of a sine.

```
self.HR = 1.16 # heart rate in beats per second; from 70 bpm
self.HR_inv = 1.0/self.HR
self.SV = 0.27
self.A1 = self.area(1)
self.flow_per_unit_area = self.volume_flow/self.A1
sin_integration_factor = 0.315
self.v_max = self.flow_per_unit_area/sin_integration_factor
```

As before, we have a global function returning the sine as a Function - object,

```
def get_sine_input_function(V, z_index, HR, HR_inv, v_max):
  v_z = "sin(2*pi*HR*fmod(t,T))*(v_max)"
  vel = ["0.0", "0.0", "0.0"]
  vel[z_index] = v_z
  class Sine(Function):
    cpparg = tuple(vel)
    defaults = {'HR}:HR, \emp{v_max}:v_max, \emp{T}:HR_inv}

  return Sine(V)
```

that is called instead of `get_pulse_input_function` in the function named `boundary_conditions`:

```
self.g1 = get_sine_input_function(V, self.z_index, self.factor, self.A, self.HR_inv, self.HR,\
                                  self.b, self.f1).
```

The pulse and the sine are sketched in Figure 27.2. Both functions are marked at the points of interest: maximum systolic flow, around the transition from systole to diastole and the (first, local) minimum. Results for sinusoidal boundary conditions are shown in Figure 27.5 The shape of the flow profile is similar in every time step, only the magnitudes change. No bidirectional flow was discovered in the transition from positive to negative flow. Compared to the results received by the more realistic pulse function, the velocity profile generated from sinusoidal boundaries is more homogeneous over the cross section.

27.3.4 Example 3. Cord Shape and Position.

According to (Loth et al., 2001), (Alperin et al., 2006), the present flow is inertia dominated, meaning that the shape of the cross section should not influence the pressure gradient. Changing the length of vectors describing the ellipse from

```
c = 0.4;
d = 0.4;
```

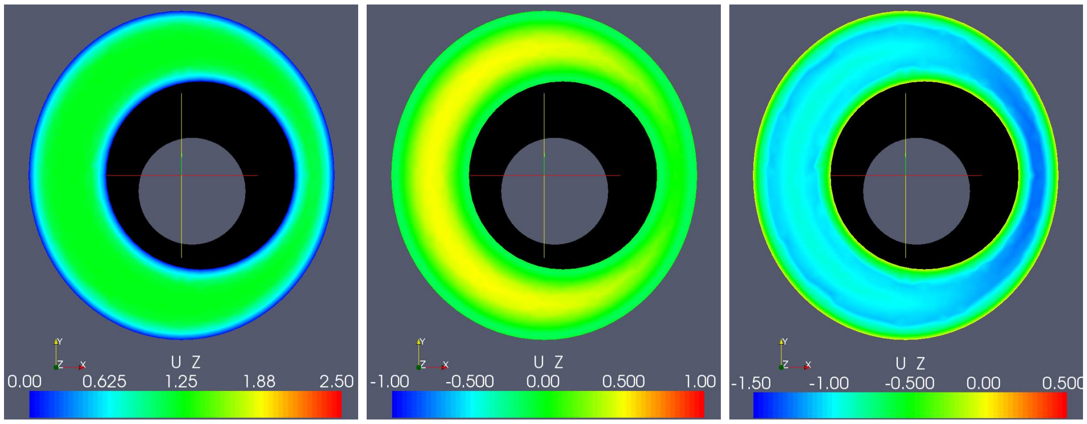


Figure 27.5: Case: Circular Cord. The velocity in z-direction as response to a sine boundary condition for the time steps $t = 0.2, 0.4, 0.6$.

.

to

```
c = 0.32;
d = 0.5;
```

transforms the cross section of the inner cylinder to an elliptic shape with preserved area. The simulation results are collected in Figure 27.6. Comparisons showed that the pressure gradient was identical for the two cases, the different shape is however reflected in the flow profiles.

A further perturbation of the SAS cross sections was achieved by changing the moving of the center of the elliptic cord from

```
move = 0.08;
```

to

```
move = 0.16;
```

Also for this case the pressure field was identical, with some variations in the flow profiles.

27.3.5 Example 4. Cord with Syrinx.

Syrinxes expand the cord so that it occupies more space of the spinal SAS. Increasing the cord radius from 4 mm to 5 mm [2] decreases the cross sectional area by almost one third to 0.64 cm^2. The resulting flow is shown in Figure 27.8. Apart from the increased velocities, we see bidirectional flow already at t = 0.18 and at

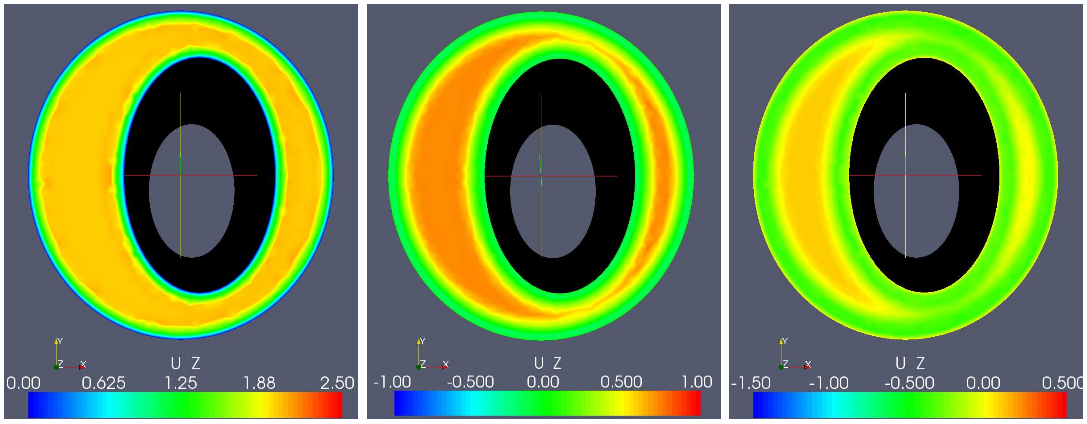


Figure 27.6: Case: Elliptic cord. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

.

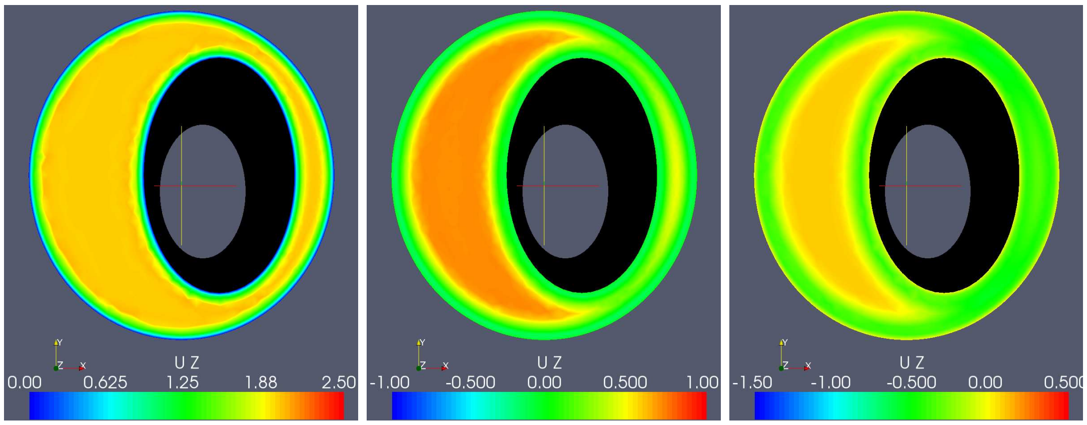


Figure 27.7: Case: Translated elliptic cord. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

.

Problem	D [3] in cm	v_{max} [4] in cm/s	Re	We
Example 1	0.54	2.3	177	17
Example 2	0.54	0.92	70	17
Example 4	0.45	3.2	205	14

Table 27.3: Characteristic values for the examples 1, 2 and 3.

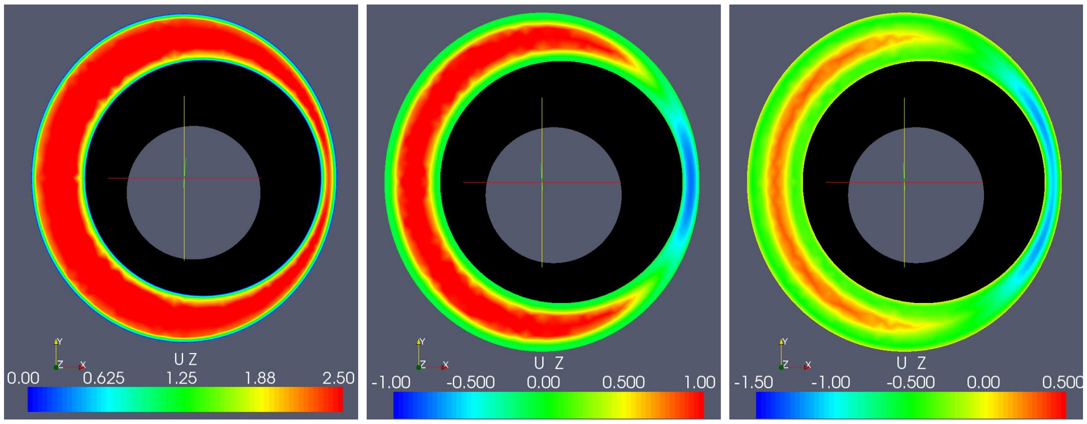


Figure 27.8: Case: Enlarged cord diameter. The velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

t=0.25 as before. The fact that diastolic back flow is visible at t = 0.18, shows that the pulse with its increased amplitude travels faster.

Comparing Reynholds and Womersly numbers shows a clear differene for the above described examples 1, 2 and 3. Example 2 is marked by a clearly lower maximum velocity at inflow and outflow boundary that leads to a rather low Reynholdsnumber. Due to the different inflow and outflow area, Example 4 has a lower Womerley number, leading to an elevated maximum velocity at the boundary and clearly increased Reynholds number. These numbers help to quantify the changes introduced by variations in the model. For the chosen model, the shape of the pulse function at the boundary condition as well as the cross sectional area have great influence on the simulation results. As earlier shown by Loth et al. (Loth et al., 2001), altering the shape of the cross sections does not seem to influence the flow greatly.

[2]which equals to set the variables c and d in the geo-file to 0.5

CHAPTER 28

Turbulent Flow and Fluid-Structure Interaction with Unicorn

By Johan Hoffman, Johan Jansson, Niclas Jansson, Claes Johnson and Murtazo Nazarov

Chapter ref: **[hoffman-1]**

28.1 Introduction

For many problems involving a fluid and a structure, decoupling the computation of the two is not possible for accurate modeling of the phenomenon at hand, instead the full fluid-structure interaction (FSI) problem has to be solved together as a coupled problem. This includes a multitude of important problems in biology, medicine and industry, such as the simulation of insect or bird flight, the human cardiovascular and respiratory systems, the human speech organ, the paper making process, acoustic noise generation in exhaust systems, airplane wing flutter, wind induced vibrations in bridges and wave loads on offshore structures. Common for many of these problems is that for various reasons they are very hard or impossible to investigate experimentally, and thus reliable computational simulation would open up for detailed study and new insights, as well as for new important design tools for construction.

Computational methods for FSI used today are characterized by a high computational cost, and a lack of generality and reliability. In particular, major open challenges of computational FSI include: (i) robustness of the fluid- structure coupling, (ii) for high Reynolds numbers the computation of turbulent fluid flow, and (iii) efficiency and reliability of the computations in the form of adaptive methods and quantitative error estimation.

The FEniCS project aims towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation, and application. The Unicorn project is a realization of this effort in the field of continuum mechanics, with a range of challenging problems that traditionally demand a number of specialized methods and codes. The basis of Unicorn is an adaptive finite element method and a unified continuum formulation, which offer new possibilities for computational modeling of high Reynolds number turbulent flow, gas dynamics and fluid-structure interaction.

Unicorn, which is based on the DOLFIN/FFC/FIAT suite, and PETSc for linear algebra, is today central in a number of applied research projects, characterized by large problems, complex geometry and constitutive models, and a need for results with quantitative error control. We here present some key elements of Unicorn and the underlying theory, and illustrate how this opens for a number of breakthroughs in applied research. The Unicorn implementation is described in detail in Chapter 20.1.

28.2 Continuum mechanics models

Continuum mechanics is based on conservation laws for mass, momentum and energy, together with constitutive laws for stresses. A Newtonian fluid is characterized by a linear relation between the viscous stress and the strain, together with a fluid pressure, resulting in the Navier-Stokes equations. Many common fluids, including water and air at subsonic velocities, can be modeled as incompressible fluids, where the pressure acts as a Langrangian multiplier enforcing a divergence free velocity. In models of gas dynamics the pressure is given from the internal energy, with an ideal gas corresponding to a linear relation. Solids and non-Newtonian fluids can be described by arbitrary complex laws relating the stress to displacements, strains and internal energies.

Newtonian fluids are characterized by two important non-dimensional numbers: the Reynolds number Re, measuring the importance of viscous effects, and the Mach number M, measuring compressibility effects by relating the fluid velocity to the speed of sound. High Re flow is characterized by partly turbulent flow, and high M flow by shocks and contact discontinuities, all phenomena associated with complex flow on a range of scales. The Euler equations corresponds to the limit of inviscid flow where $Re \to \infty$, and incompressible flow corresponds to the limit of $M \to 0$.

In this Chapter we focus on incompressible fluids, and leave a discussion on compressible fluids to Chapter **??**

28.3 Mathematics framework for fluid mechanics

The mathematical theory for the Navier-Stokes equations is incomplete, without any proof of existence or uniqueness, formulated as one of the Clay Institute $1 million problems. What is available, is the proof of existence of weak solutions by Leray from 1934, but this proof does not extend to the inviscid case of the Euler equations. No general uniqueness result is available for weak solutions, which limit the usefulness of the concept.

In (Hoffman and Johnson, 2008a), weak (output) uniqueness is introduced, to characterize well-posedness of weak solutions with respect to functionals $M(u)$ of the solution u. This framework extends to the Euler equations, and also to compressible flow. The basic result takes the form

$$|M(u) - M(U)| \leq S(\|R(u)\|_{-1} + \|R(U)\|_{-1}) \tag{28.1}$$

with $\|\cdot\|_{-1}$ a weak norm measuring residuals $R(\cdot)$ of two weak solutions u and U, and with S a stability factor given by a duality argument connecting local errors to output errors in $M(\cdot)$.

28.4 Adaptive computational fluid modeling

Computational methods in fluid mechanics are typically very specialized; for a certain range of Re or M, or for a particular class of geometries. In particular, there is a wide range of turbulence models and shock capturing techniques.

The goal of Unicorn is to design one method with one implementation, capable of modeling general geometry and the whole range of parameters Re and M. We use the mathematical framework of well-posedness as a general foundation for Newtonian fluid mechanics, where a General Galerkin (G2) finite element method offers a robust algorithm to compute weak solutions (Hoffman and Johnson, 2006a). The UFL implementation of a G2 method in Unicorn is shown in Fig.28.1.

Adaptive G2 methods developed in (Hoffman, 2005, 2006a, 2009, Hoffman and Johnson, 2006b) are based on a posteriori error estimates of the form:

$$|M(u) - M(U)| \leq \sum_K \mathcal{E}_K \tag{28.2}$$

with $M(u)$ the target output to compute and $M(U)$ the approximation, with u and U G2 solutions, and $\mathcal{E}_K$ a local error indicator for cell K. The error indicator $\mathcal{E}_K$ is constructed as from the residual, measuring local errors, weighted by the solution to a dual (adjoint) problem measuring the effect of local errors on the output $M(\cdot)$. The UFL implementation of the dual problem in Unicorn is shown in Fig.28.2

The computational mesh is then modified according to $\mathcal{E}_K$, by mesh refinement, coarsening or smoothing. The algorithms underlying the parallel implementation the adaptive algorithm is described in detail in Chapter **??**.

28.5 Turbulent boundary layers

The choice of boundary conditions at a solid wall is critical for accurate modeling of fluid flow, in particular to capture flow separation phenomena. Since full resolution of a turbulent boundary layers is out of reach, the standard way to handle the problem is to divide the computational domain into: (i) an interior part Ω, and (ii) a boundary layer. In the boundary layer a simplified model of the flow is used to provide boundary conditions to the Navier-Stokes equations to be solved in the interior part Ω. Boundary conditions may be in the form of velocities or stresses, and the coupling between (i) and (ii) may be one-way from (ii) to (i), or more closely coupled. Boundary layer models are developed based on experimental data, theory or computation (in a multiscale framework). For an overview of boundary layer modeling, see e.g. (Sagaut, 2005, Sagaut et al., 2006).

Turbulent boundary layer modeling in Unicorn is based on recent work (Hoffman, 2006c, 2009, Hoffman and Johnson, 2008b, Jansson and Hoffman, 2009) where the turbulent boundary layer is modelled by a skin friction stress at the boundary. That is, we append the Navier-Stokes equations with the following boundary conditions for the velocity u and stress σ:

$$u \cdot n = 0, \tag{28.3}$$

$$u \cdot \tau_k + \beta^{-1} n^T \sigma \tau_k = 0, \quad k = 1, 2, \tag{28.4}$$

for $(x, t) \in \Gamma_{solid} \times [0, T]$, with $n = n(x)$ an outward unit normal vector, and $\tau_k = \tau_k(x)$ orthogonal unit tangent vectors of Γ_{solid}. We use matrix notation with all vectors v being column vectors and the corresponding row vector is denoted v^T. For a tangent velocity $u \cdot \tau_k \sim 1$, the friction parameter $\beta \sim F_f$, with F_f the skin friction stress. The weak implementation of the skin friction boundary condition in Unicorn is shown in Fig.28.1.

28.6 Unified Continuum model

For robust fluid-structure interaction Unicorn is based on Unified Continuum (UC) modeling (Hoffman et al., 2009), where the combined fluid-structure continuum is described by conservation laws for mass, momentum and energy, and a stress σ and phase variable θ are kept as data for defining properties of the continuum, such as constitutive laws and material parameters. The equations are evaluated in the fixed actual (Euler or laboratory) coordinate system.

The current version of Unicorn implements an incompressible continuum, which simplifies modeling by decoupling the energy equation from conservation of mass and momentum. The extension to a compressible continuum is under way, based on the compressible solver of Unicorn, described in Chapter **??**.

We start with conservation of mass, momentum and energy, together with a convection equation for a phase function θ over a space-time domain $Q = [\Omega \times [0, T]]$ with Ω an open domain in R^3 with boundary Γ:

$$\boxed{\begin{aligned} D_t\rho + D_{x_j}(u_j\rho) &= 0 \quad \text{(Mass conservation)} \\ D_t m_i + D_{x_j}(u_j m_i) &= D_{x_j}\sigma_i \quad \text{(Momentum conservation)} \\ D_t e + D_{x_j}(u_j e) &= D_{x_j}\sigma_i u_i \quad \text{(Energy conservation)} \\ D_t\theta + D_{x_j}u_j\theta &= 0 \quad \text{(Phase convection equation)} \end{aligned}} \tag{28.5}$$

together with initial and boundary conditions, where the stress is the Cauchy (laboratory) stress and the phase variable is used to define material data such as constitutive law for the stress and material parameters. Note that in this continuum description the coordinate system is fixed (Euler).

Above an indexed Einstein notation is used with the derivative of a function f with regard to the variable x denoted as $D_x f$, and the derivative with regard to component x_i of component f_j denoted as $D_{x_i} f_j = \nabla f_j$. Repeated indices denote a sum: $D_{x_i} f_i = \sum_{i=1}^{d} D_{x_i} f_i = \nabla \cdot f$. Similarly we can express derivatives with respect to any variable: $D_u u = 1$.

For an incompressible continuum we have:

$$\begin{aligned} \rho(D_t u_i + u_j D_j u_i) &= D_{x_j}\sigma_{ij} \\ D_{x_j} u_j &= 0 \end{aligned}$$

where now the energy equation is decoupled and we can omit it. The total stress can be decomposed into constitutive and forcing stresses:

$$D_{x_j}\sigma_{ij} = D_{x_j}\sigma_{ij} + D_{x_j}\sigma_{ij}^{f} = D_{x_j}\sigma_{ij} + f_i$$

We can then pose constitutive relations between the constitutive (Cauchy) stress component σ and other variables such as the velocity u.

This continuum modeling framework is simple and compact, close to the formulation of the original conservation laws, without mappings between coordinate systems. This allows simple manipulation and processing for error estimation and implementation. It is also general, constitutive laws can be chosen to model simple or complex solids and fluids in interaction, with individual parameters.

We choose a G2 discretization of the UC, based on streamline diffusion stabilization and a local ALE map over the mesh T^h. The implementation in Unicorn is shown in Fig. **??**, and for details on the method see (Hoffman et al., 2009).

28.7 Constitutive laws

The UC model allows us to choose different constitutive laws describing the behaviour of the particular material for each phase.

28.7.1 Fluid laws

The current version of Unicorn implements only Newtonian fluids, although non-Newtonian fluids are expected to be compatible with the UC framework and the Unicorn implementation, where they could be seen as relatives of viscous and plastic solid constitutive laws as given below.

- For a fluid phase we typically choose a Newtonian law: $\sigma = 2\nu\epsilon - pI$

28.7.2 Solid laws

For a solid phase there exists a multitude of choices for constitutive laws. Several possible laws are listed below. The primitives for describing laws is the deformation gradient F and the velocity u. The main relations between u and F are summarized as:

$$\boxed{\begin{aligned} D_t F &= \nabla u\, F \\ D_t F^{-1} &= -F^{-1}\nabla u \\ B &= FF^T \end{aligned}} \qquad (28.6)$$

Using the above relation to compute F, constitutive laws can be expressed coupling the stress σ to the deformation F, typically in the form $B = FF^T$. F could also be eliminated to formulate stress rate laws only in terms of the stress σ and the velocity u. We here present some possible choices, with extension to plasticity through a stress rate law:

- A common example is a Neo-Hookean law: $\sigma = \mu B - pI$.
- Selecting the component $\sigma_D = \mu B$ and differentiating with regard to time B can be eliminated so that $D_t\sigma_D = 2\mu\epsilon(u) + \nabla u\sigma_D + \sigma_D\nabla u^\top$.
- A (compressible) elasto-plastic variant of this model is: $D_t\sigma + \nu^{-1}(\sigma - \pi\sigma) = E\epsilon(u)$, where ν is a viscosity coefficient and $\pi\sigma$ denotes the projection of σ onto a (convex) set of plastically admissible stresses.

28.8 Applications

We now illustrate the capabilities of the Unicorn solver by showing snapshots from simulations, described in detail elsewhere. The chosen simulations couple to the following challenges of computational mechanics: simulation of turbulent flow and turbulent flow separation, and robust fluid-structure interaction.

28.8.1 Turbulent separation

Viscous effects in the boundary layer is traditionally used as a mechanism to explain flow separation, not only for low Reynolds numbers Re but also for high Re where otherwise inertial effects dominate. In particular, viscous effects in the boundary layer is often presented as the resolution of the d'Alembert paradox (Stewartson, 1981), seemingly disqualifying the inviscid Euler equations with slip boundary conditions as a model for high Re flow. The significance of the boundary layer for explaining turbulent flow separation was questioned in (Hoffman and Johnson, 2008b), where instead a mechanism for inviscid separation was suggested based on exponential growth of streamwise vorticity at separation. In particular, a new resolution of the d'Alembert paradox was presented based on this instability of potential flow at separation.

In (Jansson and Hoffman, 2009) a computational study is presented using Unicorn, where the drag force of a circular cylinder is computed adaptively based on a posteriori error estimation. In particular, the phenomenon of drag crisis is targeted, characterized by a sudden drop in the non-dimensional drag coefficient for a cylinder for Re increasing beyond a critical size of about 10^5. By decreasing the skin friction parameter β, modeling an increasing Re, the drag crisis scenario is reproduced using Unicorn, in agreement with the high Re experimental data available in the literature (Zdravkovich, 2003). In particular, for vanishing skin friction the flow approaches a state independent of the skin friction parameter, which thus correspond to a free slip boundary condition, see Fig.28.4-28.6.

If indeed a slip boundary condition, without the boundary layer, is a good model for high Re flow separation, this represents a major breakthrough for turbulence simulation, which opens for new advanced simulations in aero- and hydrodynamics. In Fig.28.7-28.8 this is exemplified by modeling the turbulent flow past a NACA 0012 airfoil under increasing angle of attack, and the turbulent flow past a realistic geometry of a full car (Hoffman and Johnson, 2006a), for which Unicorn allows for time resolved simulations using the capacity of a laptop computer.

28.8.2 Robust fluid-structure interaction

A main challenge of fluid-structure interaction is the stability of the fluid-structure coupling. The unified continuum model of Unicorn provide a monolitic method which is robust, and also allows for a flexible constitutive modeling. As a special case of the FSI solver, Unicorn also offers a robust solver for flow problems in deforming domains.

There is a multitude of areas in which fluid-structure interaction plays an important role, not the least in biomedicine that poses a number of complex fluid-structure interaction problems. One such challenge is the human heart, where cardiac muscle contracts to pump the blood through the cardiovascular system. Depending on the context various computational models of the heart can be constructed. To study the fluid dynamics of the blood inside the heart, one can reconstruct the deformation of the heart from medical imaging, to be used as basis for a deforming domain fluid dynamics model of the blood flow. This is the approach underlying the simulation in Fig.28.9, where the blood flow in the left ventricle is modeled using Unicorn. The blood is here assumed to be incompressible, and the deforming domain is based on patient specific medical image data. See (Aechtner, 2009) for more details on this model.

Unicorn is designed to be able to handle large structure deformations interacting with complex fluid flow. In Fig.28.10 we present a model problem of a flexible structure interacting with turbulent flow in 3D, in the form of a fixed cube in high Re flow with a thin flexible flag mounted in the downstream wake. Violent bending and torsion motion along the long axis of the flag is observed, and we note that the method is robust for these large structure deformations and highly fluctuating flow.

```
scalar = FiniteElement("Lagrange", tetrahedron, 1)
vector = VectorElement("Lagrange", tetrahedron, 1)
constant_scalar = FiniteElement("Discontinuous Lagrange", tetrahedron, 0)
constant_vector = VectorElement("Discontinuous Lagrange", tetrahedron, 0)

v      = TestFunction(vector)      # test basis function
U      = TrialFunction(vector)     # trial basis function
um     = Function(vector)          # cell mean linearized velocity
u0     = Function(vector)          # velocity from previous time step
f      = Function(vector)          # force term
p      = Function(scalar)          # pressure
delta1 = Function(constant_scalar) # stabilization parameter
delta2 = Function(constant_scalar) # stabilization parameter
tau_1  = Function(vector)          # force term
tau_2  = Function(vector)          # force term
beta  = Function(scalar)           # friction parameter

k  = Constant(tetrahedron) # time step
nu = Constant(tetrahedron) # viscosity

i0 = Index()    # index for tensor notation
i1 = Index()    # index for tensor notation
i2 = Index()    # index for tensor notation

# Galerkin discretization of bilinear form
G_a  = (inner(v, U) + k*nu*0.5*inner(grad(v), grad(U)) \
    + 0.5*k*v[i0]*um[i1]*U[i0].dx(i1))*dx \
    + 0.5*k*beta*(inner(U,tau_1)*inner(v,tau_1) \
    + inner(U,tau_2)*inner(v,tau_2))*ds

# Least squares stabilization of bilinear form
SD_a = (delta1*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*U[i0].dx(i2) \
           + delta2*k*0.5*div(v)*div(U))*dx

# Galerkin discretization of linear form
G_L  = (inner(v, u0) + k*inner(v, f) + k*div(v)*p \
          - k*nu*0.5*inner(grad(v), grad(u0)) \
          - 0.5*k*v[i0]*um[i1]*u0[i0].dx(i1))*dx \
          - 0.5*k*beta*(inner(u0,tau_1)*inner(v,tau_1) \
          + inner(u0,tau_2)*inner(v,tau_2))*ds

# Least squares stabilization of linear form
SD_L = (- delta1*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*u0[i0].dx(i2) \
           - delta2*k*0.5*div(v)*div(u0))*dx

# Bilinear and linear forms
a = G_a + SD_a
L = G_L + SD_L
```

Figure 28.1: Source code for bilinear and linear forms for solving the incompressible Navier-Stokes equations.

```
scalar = FiniteElement("Lagrange", tetrahedron, 1)
vector = VectorElement("Lagrange", tetrahedron, 1)
constant_scalar = FiniteElement("Discontinuous Lagrange", tetrahedron, 0)
constant_vector = VectorElement("Discontinuous Lagrange", tetrahedron, 0)

v      = TestFunction(vector)      # test basis function
U      = TrialFunction(vector)     # trial basis function
um     = Function(vector)          # primal velocity
u0     = Function(vector)          # velocity from previous time step
f      = Function(vector)          # force term
p      = Function(scalar)          # pressure
delta1 = Function(constant_scalar) # stabilization parameter
delta2 = Function(constant_scalar) # stabilization parameter

k  = Constant(cell) # time step
nu = Constant(cell) # viscosity

up     = Function(vector) # cell mean linearized primal velocity

i0 = Index()    # index for tensor notation
i1 = Index()    # index for tensor notation
i2 = Index()    # index for tensor notation

# Galerkin discretization of bilinear form
G_a  = (inner(v, U) + k*nu*0.5*inner(grad(v), grad(U)) \
          - 0.5*k*v[i0]*up[i1]*U[i0].dx(i1) \
          + 0.5*k*v[i0]*up[i1].dx(i0)*U[i1])*dx
# Least squares stabilization of bilinear form
SD_a = (delta1*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*U[i0].dx(i2) \
           + delta1*k*0.5*up[i1].dx(i0)*v[i1]*up[i2].dx(i0)*U[i2] \
           + delta2*k*0.5*div(v)*div(U))*dx

# Galerkin discretization of linear form
G_L  = (inner(v, u0) + k*inner(v, f) + k*div(v)*p \
          - k*nu*0.5*inner(grad(v), grad(u0)) \
          + 0.5*k*v[i0]*up[i1]*u0[i0].dx(i1) \
          - 0.5*k*v[i0]*up[i1].dx(i0)*u0[i1])*dx
# Least squares stabilization of linear form
SD_L = (- delta1*k*0.5*um[i1]*v[i0].dx(i1)*um[i2]*u0[i0].dx(i2) \
           - delta1*k*0.5*up[i1].dx(i0)*v[i1]*up[i2].dx(i0)*u0[i2] \
           - delta2*k*0.5*div(v)*div(u0))*dx

# Bilinear and linear forms
a = G_a + SD_a
L = G_L + SD_L
```

Figure 28.2: Source code for bilinear and linear forms for solving the dual incompressible Navier-Stokes equations.

```
def tomatrix(q):
    return [ [q[d * i + j] for i in range(d)] for j in range(d) ]

def ugradu(u, v):
    return [dot(u, grad(v[i])) for i in range(d)]

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

def E(e, mu, lmbda):
    Ee = mult(2.0 * mu, e) + mult(lmbda, mult(trace(e), Identity(d)))
    return Ee

UPale = UP - WP
UPalem = UPm - WPm

sigmaM = tomatrix(sigma)

Sf = mult(P, Identity(d)) - mult(nu, grad(UP))
Ss = mult(P, Identity(d)) - mult(1.0, sigmaM)
S = mult(phi, Sf) + mult(1.0 - phi, Ss)

def f(u, v):
    return -(dot(ugradu(UPale, u), v) - dot(S, grad(v))) + \
        -dot(mult(d2, div(u)), div(v)) + \
        -mult(d1, dot(ugradu(UPalem, u), ugradu(UPalem, v))) + \
        dot(mult(1.0 - phi, ff), v)

def dfdu(u, k, v):
    return -dot(ugradu(UPale, u), v) + \
        -mult(1 - phi, mult(k, dot(E(epsilon(u), mu, lmbda), grad(v)))) + \
        -mult(phi, mult(nu, dot(grad(u), grad(v)))) + \
        -mult(d2, dot(div(u), div(v))) + \
        -mult(d1, dot(ugradu(UPalem, u), ugradu(UPalem, v)))

# cG(1)
def F(u, u0, k, v):
    uc = mult(0.5, u + u0)
    return (-dot(u, v) + dot(u0, v) + mult(k, f(uc, v)))

def dFdu(u, u0, k, v):
    uc = mult(0.5, u)
    return (-dot(u, v) + mult(k, dfdu(uc, k, v)))

a = (dFdu(U1, U0, k, v)) * dx
L = (dFdu(UP, U0, k, v) - F(UP, U0, k, v)) * dx
```

Figure 28.3: Source code for bilinear and linear forms for a unified continuum model.

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

Figure 28.4: Turbulent flow separation (Jansson and Hoffman, 2009): velocity vectors at surface of cylinder; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

Figure 28.5: Turbulent flow separation (Jansson and Hoffman, 2009): pressure isosurfaces; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

Figure 28.6: Turbulent flow separation (Jansson and Hoffman, 2009): velocity streamlines; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

missing figure to be added *missing figure to be added* *missing figure to be added*

missing figure to be added *missing figure to be added* *missing figure to be added*

missing figure to be added *missing figure to be added* *missing figure to be added*

Figure 28.7: Flow past NACA 0012 airfoil (Hoffman and Johnson, 2006a): velocity magnitude (upper), pressure (middle), and non-transversal vorticity (lower), for angles of attack 4, 10, 18°.

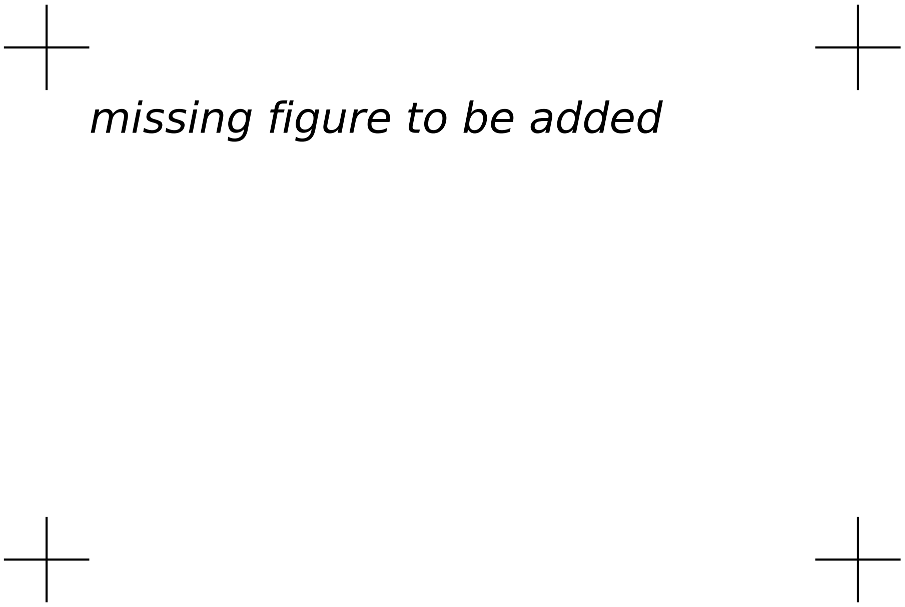


Figure 28.8: Streamlines of turbulent flow around a car (simulations by Murtazo Nazarov, with geometry by courtesy of Volvo Cars).

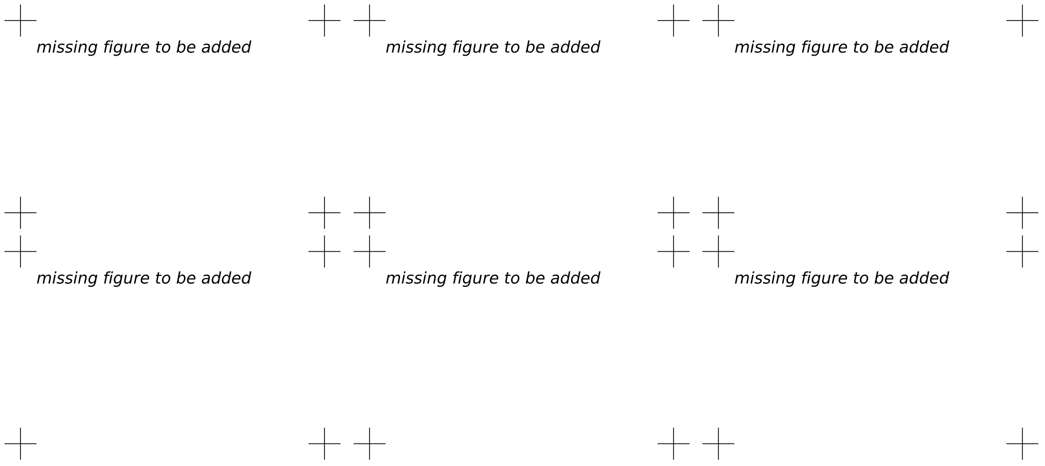

missing figure to be added
missing figure to be added
missing figure to be added
missing figure to be added
missing figure to be added
missing figure to be added

Figure 28.9: Blood flow simulation of the left ventricle of a human heart: snapshots of surface pressure (upper) and velocity (lower). The geometrical model is constructued by Ulf Gustafsson och Per Vesterlund at Umeå Universty, and the simulations performed by Matthias Aechtner at KTH (Aechtner, 2009).

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

missing figure to be added *missing figure to be added*

Figure 28.10: Simulation of turbulent flow past a square cylinder with an elastic flag attached downstream (Hoffman et al., 2009): plot of cut of the mesh, isosurface of pressure and fluid-structure phase interface. Going from initial state top left to illustrating violent bending and torsion motion along the long axis of the flag.

CHAPTER 29

Fluid–Structure Interaction using Nitsche's Method

By Kristoffer Selim and Anders Logg

Chapter ref: **[selim]**

In this study, we present a 2D fluid–structure interaction (FSI) simulation of a channel flow containing an elastic valve that may undergo large deformations. The FSI occurs when the fluid interacts with the solid structure of the valve, exerting pressure that causes deformation of the valve and, thus, alters the flow of the fluid itself. To describe the equations governing the fluid flow and the elastic valve, two separate meshes are used and Nitsche's method is used to couple the equations on the two meshes. The method is based on continuous piecewise approximations on each mesh with weak enforcement of the proper continuity at the interface defined by the boundary of one of the overlapping meshes.

CHAPTER 30

Improved Boussinesq Equations for Surface Water Waves

By Nuno D. Lopes, P. J. S. Pereira and L. Trabucho

Chapter ref: **[lopes]**

▶ Editor note: *Move macros to common macros after deciding what to do about bold fonts.*

▶ Author note: *We have replaced bold fonts with vector notation (e.g.:* u *changed to* $\vec{u}$*).*

▶ Editor note: *List authors with full names*

The main motivation of this work is the implementation of a general solver for some of the improved Boussinesq models. Here, we use the second order model proposed by Zhao et al. (Zhao et al., 2004) to investigate the behaviour of surface water waves. Some effects like surface tension, dissipation and wave generation by natural phenomena or external physical mechanisms are also included. As a consequence, some modified dispersion relations are derived for this extended model.

30.1 Introduction

The FEniCS project, via DOLFIN and FFC, provides a good support for the implementation of large scale industrial models. We implement a solver for some of the Boussinesq type systems to model the evolution of surface water waves in a variable depth seabed. This type of models is used, for instance, in harbour simulation[1], tsunami generation and propagation as well as in coastal dynamics.

[1]See Fig. 30.1 for an example of a standard harbour.

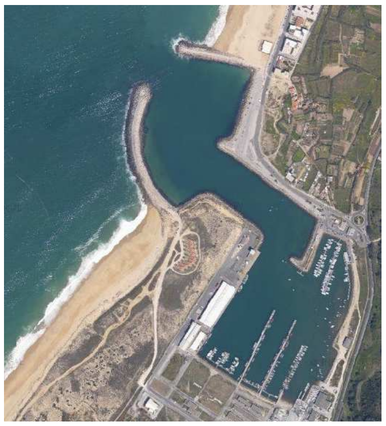


Figure 30.1: Nazaré's harbour, Portugal.

► <u>Editor note</u>*: Need to get permission for this figure!*

There are several **B**oussinesq models and some of the most widely used are those based on the wave **E**levation and horizontal **V**elocities formulation (BEV) (see, e.g., (Liu and Woo, 2004), (Walkley and Berzins, 2002)).

In the next section the governing equations for surface water waves are presented. From these equations different types of models can be derived. We consider only the wave **E**levation and velocity **P**otential (BEP) formulation. Thus, the number of system equations is reduced when compared to the BEV models. Two different types of BEP models are taken into account:

i) a standard sixth-order model (see subsection 30.2.1);

ii) the second-order model proposed by **Z**hao, **T**eng and **C**heng (ZTC) (Zhao et al., 2004) (see subsection 30.2.2).

We use the sixth-order model to illustrate a standard technique in order to derive a Boussinesq-type model. In the subsequent sections, only the ZTC model is considered. Note that these two models are complemented with some extra terms, due to the inclusion of effects like dissipation, surface tension and wave generation by moving an impermeable bottom or using a source function.

An important characteristic of the modified ZTC model, including dissipative effects, is presented in the third section, namely, the dispersion relation.

In the fourth and fifth sections, we describe several types of wave generation, absorption and reflection mechanisms. Initial conditions for a solitary wave and a periodic wave induced by Dirichlet boundary conditions are also presented. Moreover, we complement the ZTC model using a source function to generate surface water waves, as proposed in (Wei et al., 1999). Total reflective walls are modelled by standard zero Neumann conditions for the surface elevation and velocity potential. The wave energy absorption is simulated using sponge layers.

The following section is dedicated to the numerical methods used in the discretization of the variational formulation. The discretization of the spatial variables is accomplished with low order Lagrange finite elements whereas the time integration is implemented using Runge-Kutta and Predictor-Corrector algorithms.

In the seventh section, the ZTC numerical model is used to simulate the evolution of a periodic wave in an harbour geometry like that one represented in Fig. 30.1.

30.2 Model derivation

As usual we consider the following set of equations for the irrotational flow of an incompressible and inviscid fluid:

$$\begin{cases} \dfrac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla_{xyz})\vec{u} = -\nabla_{xyz}\left(\dfrac{P}{\rho} + g\,z\right), \\ \nabla_{xyz}\times\vec{u} = \vec{0}, \\ \nabla_{xyz}\cdot\vec{u} = 0, \end{cases} \tag{30.1}$$

where $\vec{u}$ is the velocity vector field of the fluid, P the pressure, g the gravitational acceleration, ρ the mass per unit volume, t the time and the differential operator $\nabla_{xyz} = \left[\dfrac{\partial}{\partial x}, \dfrac{\partial}{\partial y}, \dfrac{\partial}{\partial z}\right]$. A Cartesian coordinate system is adopted with the horizontal x and y-axes on the still water plane and the z-axis pointing vertically upwards (see Fig. 30.2). The fluid domain is bounded by the bottom seabed at $z = -h(x,y,t)$ and the free water surface at $z = \eta(x,y,t)$. In Fig. 30.2, L, A and

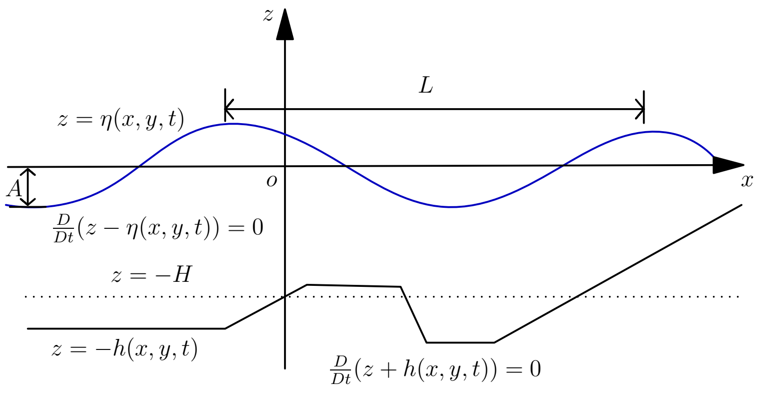

z
L
$z = \eta(x,y,t)$
A
o
x
$\frac{D}{Dt}(z-\eta(x,y,t)) = 0$
$z = -H$
$z = -h(x,y,t)$
$\frac{D}{Dt}(z+h(x,y,t)) = 0$

Figure 30.2: Cross-section of the water wave domain.

H are the characteristic wave length, wave amplitude and depth, respectively. Note that the material time derivative is denoted by $\frac{D}{Dt}$.

From the irrotational assumption (see $(30.1)_2$), one can introduce a velocity potential function, $\phi(x, y, z, t)$, to obtain Bernoulli's equation:

$$\frac{\partial \phi}{\partial t} + \frac{1}{2}\nabla_{xyz}\phi \cdot \nabla_{xyz}\phi + \frac{P}{\rho} + g\,z = f(t), \tag{30.2}$$

where $f(t)$ stands for an arbitrary function of integration. Note that one can remove $f(t)$ from equation (30.2) if ϕ is redefined by $\phi + \int f(t)\,\mathrm{d}t$. From the incompressibility condition (see $(30.1)_3$) the velocity potential satisfies Laplace's equation:

$$\nabla^2\phi + \frac{\partial^2 \phi}{\partial z^2} = 0, \tag{30.3}$$

where ∇ is the horizontal gradient operator given by $\nabla = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right]$. To close this problem, the following boundary conditions must be satisfied:

i) the kinematic boundary condition for the free water surface:

$$\frac{\partial \phi}{\partial z} = \frac{\partial \eta}{\partial t} + \nabla\phi \cdot \nabla\eta, \quad z = \eta; \tag{30.4}$$

ii) the kinematic boundary condition for the impermeable sea bottom:

$$\frac{\partial \phi}{\partial z} + (\nabla\phi \cdot \nabla h) = -\frac{\partial h}{\partial t}, \quad z = -h; \tag{30.5}$$

iii) the dynamic boundary condition for the free water surface:

$$\frac{\partial \phi}{\partial t} + g\eta + \frac{1}{2}\left(|\nabla\phi|^2 + \left(\frac{\partial \phi}{\partial z}\right)^2\right) + D(\phi) - W(\eta) = 0, \quad z = \eta, \tag{30.6}$$

where $D(\phi)$ is a dissipative term (see, e.g., the work by Duthyk and Dias (Dutykh and Dias, 2007)). We assume that this dissipative term is of the following form:

$$D(\phi) = \nu\frac{\partial^2 \phi}{\partial z^2}, \tag{30.7}$$

with $\nu = \bar{\mu}/\rho$ and $\bar{\mu}$ an eddy-viscosity coefficient. Note that a non-dissipative model means that there is no energy loss. This is not acceptable from a physical point of view, since any real flow is accompanied by energy dissipation.

In equation (30.6), $W(\eta)$ is the surface tension term given by:

$$W(\eta) = T\frac{\left(1 + \left(\frac{\partial \eta}{\partial y}\right)^2\right)\frac{\partial^2 \eta}{\partial x^2} + \left(1 + \left(\frac{\partial \eta}{\partial x}\right)^2\right)\frac{\partial^2 \eta}{\partial y^2} - 2\frac{\partial \eta}{\partial x}\frac{\partial \eta}{\partial y}\frac{\partial^2 \eta}{\partial x \partial y}}{(1 + |\nabla\eta|^2)^{3/2}}, \tag{30.8}$$

where T is the surface tension coefficient.

Using Laplace's equation (see (30.3)) it is possible to rewrite (30.7) as $D(\phi) = -\nu\nabla^2\phi$. Throughout the literature, analogous terms were added to the kinematic and dynamic conditions to absorb the wave energy near the boundaries. These terms are related with the sponge or damping layers and, as we will see later, they can be used to modify the dispersion relations. In addition, the linearization of equation (30.8) results in $W(\eta) = T\nabla^2\eta$. The surface tension effects are important if short waves are considered. Although the long wave assumption is made to derive these extended models, waves of short length are generated in the domain due to the waves interaction. Thus, the inclusion of surface tension in the small amplitude long wave models may be relevant. On the other hand, it is worth to mention that one of the main goals of the scientific research on Boussinesq wave models is the improvement of the range of applicability in terms of the water-depth/wave-length relationship. We refer the works by Wang et al. (Wang et al., 2008) as well as Dash and Daripa (Dash and Daripa, 2002), which included surface tension effects in the KdV (Korteweg-de Vries) and Boussinesq equations.

A more detailed description of the above equations is found in G. B. Whitham's reference book on waves (Whitham, 1974), or in the more recent book by R. S. Johnson (Johnson, 1997).

30.2.1 Standard models

In this subsection, we present a generic Boussinesq system using the velocity potential formulation. To transform equations (30.2)-(30.8) in a dimensionless form, the following scales are introduced:

$$(x', y') = \frac{1}{L}(x, y), \quad z' = \frac{z}{H}, \quad t' = \frac{t\sqrt{gH}}{L}, \quad \eta' = \frac{\eta}{A}, \quad \phi' = \frac{H\phi}{AL\sqrt{gH}}, \quad h' = \frac{h}{H}, \tag{30.9}$$

together with the small parameters

$$\mu = \frac{H}{L}, \quad \varepsilon = \frac{A}{H}. \tag{30.10}$$

In the last equation, μ is usually called the long wave parameter and ε the small amplitude wave parameter. Note that ε is related with the nonlinear terms and μ with the dispersive terms. For simplicity, in what follows, we drop the prime notation.

The Boussinesq approach consists in reducing a 3**D** problem to a 2**D** one. This may be accomplished by expanding the velocity potential in a Taylor power series in terms of z. Using Laplace's equation, in a dimensionless form, one can obtain the following expression for the velocity potential:

$$\phi(x,y,z,t)=\sum_{n=0}^{+\infty}\left((-1)^n\frac{z^{2n}}{(2n)!}\mu^{2n}\nabla^{2n}\phi_0(x,y,t)+(-1)^n\frac{z^{2n+1}}{(2n+1)!}\mu^{2n}\nabla^{2n}\phi_1(x,y,t)\right), \tag{30.11}$$

with

$$\phi_0=\phi\mid_{z=0},\quad \phi_1=\left(\frac{\partial\phi}{\partial z}\right)\mid_{z=0}. \tag{30.12}$$

From asymptotic expansions, successive approximation techniques and the kinematic boundary condition for the sea bottom, it is possible to write ϕ_1 in terms of ϕ_0 (cf. (Chen and Liu, 1994), (Zhao et al., 2004)). In this work, without loss of generality, we assume that the dispersive and nonlinear terms are related by the following equation:

$$\frac{\varepsilon}{\mu^2}=O(1). \tag{30.13}$$

Note that the Ursell number is defined by $U_r=\dfrac{\varepsilon}{\mu^2}$.

A sixth-order model is obtained if ϕ_1 is expanded in terms of ϕ_0 and all terms up to $O(\mu^8)$ are retained. Thus, the asymptotic kinematic and dynamic boundary conditions for the free water surface are rewritten as follows [2]:

$$\begin{cases}\dfrac{\partial\eta}{\partial t}+\varepsilon\nabla\cdot(\eta\nabla\phi_0)-\dfrac{1}{\mu^2}\phi_1+\dfrac{\varepsilon^2}{2}\nabla\cdot(\eta^2\nabla\phi_1)=O(\mu^6),\\ \dfrac{\partial\phi_0}{\partial t}+\varepsilon\eta\dfrac{\partial\phi_1}{\partial t}+\eta+\dfrac{\varepsilon}{2}\left|\nabla\phi_0\right|^2+\varepsilon^2\nabla\phi_0\cdot\eta\nabla\phi_1-\\ \qquad -\varepsilon^2\eta\nabla^2\phi_0\phi_1+\dfrac{\varepsilon}{2\mu^2}\phi_1^2+D(\phi_0,\phi_1)-W(\eta)=O(\mu^6),\end{cases} \tag{30.14}$$

where ϕ_1 is given by:

$$\begin{aligned}\phi_1=&-\mu^2\nabla\cdot(h\nabla\phi_0)+\frac{\mu^4}{6}\nabla\cdot\left(h^3\nabla^3\phi_0\right)-\frac{\mu^4}{2}\nabla\cdot\left(h^2\nabla^2\cdot(h\nabla\phi_0)\right)-\\&-\frac{\mu^6}{120}\nabla\cdot\left(h^5\nabla^5\phi_0\right)+\frac{\mu^6}{24}\nabla\cdot\left(h^4\nabla^4\cdot(h\nabla\phi_0)\right)+\frac{\mu^6}{12}\nabla\cdot\left(h^2\nabla^2\cdot\left(h^3\nabla^3\phi_0\right)\right)-\\&-\frac{\mu^6}{4}\nabla\cdot\left(h^2\nabla^2\cdot\left(h^2\nabla^2\cdot(h\nabla\phi_0)\right)\right)-\frac{\mu^2}{\varepsilon}\frac{\partial h}{\partial t}-\frac{\mu^2}{\varepsilon}\frac{\mu^2}{2}\nabla\cdot\left(h^2\nabla\frac{\partial h}{\partial t}\right)+\\&+\frac{\mu^2}{\varepsilon}\frac{\mu^4}{24}\nabla\cdot\left(h^4\nabla^3\frac{\partial h}{\partial t}\right)-\frac{\mu^2}{\varepsilon}\frac{\mu^4}{4}\nabla\cdot\left(h^2\nabla^2\left(h^2\nabla\frac{\partial h}{\partial t}\right)\right)+O(\mu^8).\end{aligned} \tag{30.15}$$

To obtain equation (30.15), we assume that $\dfrac{\partial h}{\partial t}=O(\varepsilon)$ (cf. (Dutykh and Dias, 2007)).

[2]Note that D and W are, now, dimensionless functions.

30.2.2 Second-order model

The low order equations are obtained, essentially, via the slowly varying bottom assumption. In particular, only $O(h, \nabla h)$ terms are retained. Also, only low order nonlinear terms, $O(\varepsilon)$, are admitted. In fact, the modified ZTC model is written retaining only $O(\varepsilon, \mu^4)$ terms.

Under these conditions, (30.14) and (30.15) lead to:

$$\begin{cases} \dfrac{\partial \eta}{\partial t} + \varepsilon \nabla \cdot (\eta \nabla \phi_0) - \dfrac{1}{\mu^2}\phi_1 = O(\mu^6), \\ \dfrac{\partial \phi_0}{\partial t} + \eta + \dfrac{\varepsilon}{2}\left|\nabla \phi_0\right|^2 - \nu^* \varepsilon \nabla^2 \phi_0 - T^* \mu^2 \nabla^2 \eta = O(\mu^6), \end{cases} \tag{30.16}$$

where $\nu^* = \nu\sqrt{H}/(AL\sqrt{g})$, $T^* = T/(gH^2)$ and

$$\phi_1 = -\mu^2 \nabla \cdot (h \nabla \phi_0) + \frac{\mu^4}{6}\nabla \cdot \left(h^3 \nabla^3 \phi_0\right) - \frac{\mu^4}{2}\nabla \cdot \left(h^2 \nabla^2 \cdot (h \nabla \phi_0)\right) - \\ - \frac{2\mu^6}{15} h^5 \nabla^6 \phi_0 - 2\mu^6 h^4 \nabla h \cdot \nabla^5 \phi_0 - \frac{\mu^2}{\varepsilon}\frac{\partial h}{\partial t} + O(\mu^8). \tag{30.17}$$

Thus, these extended equations, in terms of the dimensional variables, are written as follows:

$$\begin{cases} \dfrac{\partial \eta}{\partial t} + \nabla \cdot [(h+\eta)\nabla \Phi] - \dfrac{1}{2}\nabla \cdot [h^2 \nabla \dfrac{\partial \eta}{\partial t}] + \dfrac{1}{6}h^2 \nabla^2 \dfrac{\partial \eta}{\partial t} - \dfrac{1}{15}\nabla \cdot [h \nabla (h \dfrac{\partial \eta}{\partial t})] = -\dfrac{\partial h}{\partial t}, \\ \dfrac{\partial \Phi}{\partial t} + \dfrac{1}{2}|\nabla \Phi|^2 + g\eta - \dfrac{1}{15} g h \nabla \cdot (h \nabla \eta) - \nu \nabla^2 \Phi - T \nabla^2 \eta = 0, \end{cases} \tag{30.18}$$

where Φ is the transformed velocity potential given by:

$$\Phi = \phi_0 + \frac{h}{15}\nabla \cdot (h \nabla \phi_0). \tag{30.19}$$

The transformed velocity potential is used with two main consequences (cf. (Zhao et al., 2004)):

i) the spatial derivation order is reduced to the second order;

ii) linear dispersion characteristics, analogous to the fourth-order BEP model proposed by Liu and Woo (Liu and Woo, 2004) and the third-order BEV model developed by Nwogu (Nwogu, 1993), are obtained.

30.3 Linear dispersion relation

One of the most important properties of a water wave model is described by the linear dispersion relation. From this relation we can deduce the phase velocity, group velocity and the linear shoaling. The dispersion relation provides a good method to characterize the linear properties of a wave model. This is achieved using the linear wave theory of Airy.

In this section we follow the work by Duthyk and Dias (Dutykh and Dias, 2007). Moreover, we present a generalized version of the dispersion relation for the ZTC model with the dissipative term mentioned above. One can also include other damping terms, which are usually used in the sponge layers.

For simplicity, a **1D**-Horizontal model is considered. To obtain the dispersion relation, a standard test wave is assumed:

$$\begin{cases} \eta(x,t) = a\, e^{i(kx-\omega t)}, \\ \Phi(x,t) = -b\, i\, e^{i(kx-\omega t)}, \end{cases} \tag{30.20}$$

where a is the wave amplitude, b the potential magnitude, $k = 2\pi/L$ the wave number and ω the angular frequency. This wave, described by equations (30.20), is the solution of the linearized ZTC model, with a constant depth bottom and an extra dissipative term, if the following equation is satisfied:

$$\omega^2 - ghk^2 \frac{1 + (1/15)(kh)^2}{1 + 2/5(kh)^2} + i\nu\omega k^2 = 0. \tag{30.21}$$

Using Padé's [2,2] approximant, the dispersion relation given by last equation is accurate up to $O((kh)^4)$ or $O(\mu^4)$ when compared with the following equation:

$$\omega^2 - ghk^2 \frac{\tanh(kh)}{kh} + i\nu\omega k^2 = 0. \tag{30.22}$$

In fact, equation (30.22) is the dispersion relation of the full linear problem.

From (30.21), the phase velocity, $C = \frac{w}{k}$, for this dissipative ZTC model is given by:

$$C = -\frac{i\nu k}{2} \pm \sqrt{-\left(\frac{\nu k}{2}\right)^2 + gh\frac{(1 + 1/15(kh)^2)}{(1 + 2/5(kh)^2)}}. \tag{30.23}$$

In Fig. 30.3, we can see the positive real part of $\left(C/\sqrt{gh}\right)$ as a function of kh for the following models: full linear theory (FL), Zhao et al. (ZTC), full linear theory with a dissipative model (FL_D) and the improved ZTC model with the dissipative term (ZTC_D). From Fig. 30.3, one can also see that these two dissipative models admit critical wave numbers k_1 and k_2, such that the positive part of $Re\left(C/\sqrt{gh}\right)$ is zero for $k \geq k_1$ and $k \geq k_2$. To avoid some numerical instabilities one can optimize the ν values in order to reduce the short waves propagation.

In general, to improve the dispersion relation one can also use other transformations like (30.19), or evaluate the velocity potential at $z = -\sigma h$ ($\sigma \in [0,1]$) instead of $z = 0$ (cf. (Bingham et al., 2008), (Madsen and Agnon, 2003) and (Madsen et al., 2003)).

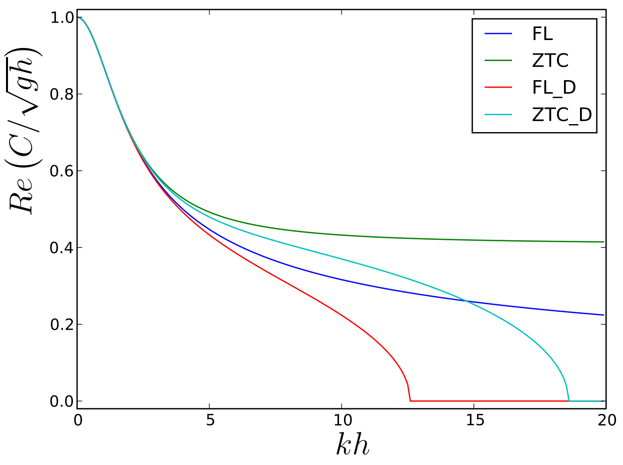


Figure 30.3: Positive part of $Re\left(C/\sqrt{gh}\right)$ as a function of kh for several models.

30.4 Wave generation

In this section some of the physical mechanisms to induce surface water waves are presented. We note that the moving bottom approach is useful for wave generation due to seismic activities. However, some physical applications are associated with other wave generation mechanisms. For simplicity, we only consider mechanisms to generate surface water waves along the x direction.

30.4.1 Initial condition

The simplest way of inducing a wave into a certain domain is to consider an appropriate initial condition. An useful and typical benchmark case is to assume a solitary wave given by:

$$\eta(x,t) = a_1 \operatorname{sech}^2(kx-\omega t) + a_2 \operatorname{sech}^4(kx-\omega t), \tag{30.24}$$

$$u(x,t) = a_3 \operatorname{sech}^2(kx-\omega t), \tag{30.25}$$

where the parameters a_1 and a_2 are the wave amplitudes and a_3 is the magnitude of the velocity in the x direction. As we use a potential formulation, Φ is given by:

$$\Phi(x,t) = -\frac{2a_3\, e^{2\omega t}}{k\left(e^{2\omega t}+e^{2kx}\right)} + K_1(t), \tag{30.26}$$

where $K_1(t)$ is a time-dependent function of integration.

In (Walkley, 1999) and (Wei and Kirby, 1995) the above solitary wave was presented as a solution of the extended Nwogu's Boussinesq model.

30.4.2 Incident wave

For time-dependent wave generation, it is possible to consider waves induced by a boundary condition. This requires that the wave surface elevation and the velocity potential must satisfy appropriated boundary conditions, e.g., Dirichlet or Neumann conditions.

The simplest case is to consider a periodic wave given by:

$$\eta(x,t) = a\sin(kx - \omega t) \tag{30.27}$$

$$\Phi(x,t) = -\frac{c}{k}\cos(kx - \omega t) + K_2(t), \tag{30.28}$$

where c is the wave velocity magnitude and $K_2(t)$ is a time-dependent function of integration. This function $K_2(t)$ must satisfy the initial condition of the problem. In equations (30.27) as well as (30.28), one can note that the parameters a, c, k and ω are not all arbitrary, since they are related by the dispersion relation. One can also consider the superposition of water waves as solutions of the full linear problem with a constant depth.

30.4.3 Source function

In the work by Wei et al. (Wei et al., 1999), a source function for the generation of surface water waves was derived. This source function was obtained, using Fourier transform and Green's functions, to solve the linearized and non homogeneous equations of the Peregrine (Peregrine, 1967) and Nwogu's (Nwogu, 1993) models. This mathematical procedure can also be adapted here to deduce the source function.

We consider a monochromatic Gaussian wave generated by the following function:

$$S(x,t) = D^{*}\exp(-\beta(x - x_s)^2)\cos(\omega t), \tag{30.29}$$

with D^{*} given by:

$$D^{*} = \frac{\sqrt{\beta}}{\omega\sqrt{\pi}} a\exp(\frac{k^2}{4\beta})\frac{2}{15}h^3k^3g. \tag{30.30}$$

In the above expressions x_s is the center line of the source function and β is a parameter associated with the width of the generation band (cf. (Wei et al., 1999)).

30.5 Reflective walls and sponge layers

Besides the incident wave boundaries where the wave profiles are given, one must close the system with appropriate boundary conditions. We consider two more types of boundaries:

i) full reflective boundaries;

ii) partial reflective or absorbing boundaries.

The first case is modelled by the following equations:

$$\frac{\partial \Phi}{\partial \vec{n}} = 0, \qquad \frac{\partial \eta}{\partial \vec{n}} = 0, \tag{30.31}$$

where $\vec{n}$ is the outward unit vector normal to the computational domain Ω. We denote Γ as the boundary of Ω.

Note that in the finite element formulation, the full reflective boundaries (equations (30.31)) are integrated by considering zero Neumann-type boundary conditions.

Coupling the reflective case and an extra artificial layer, often called sponge or damping layer, we can simulate partial reflective or full absorbing boundaries. In this way, the reflected energy can be controlled. Moreover, one can prevent unwanted wave reflections and avoid complex wave interactions. It is also possible to simulate effects like energy dissipation by wave breaking.

In fact, a sponge layer is a subset Ω_S of Ω where some extra viscosity term is added. As mentioned above, the system of equations can incorporate several extra damping terms, like that one provided by the inclusion of a dissipative model. Thus, the viscosity coefficient ν can be described by a function of the following form:

$$\nu(x,y) = \begin{cases} 0, & (x,y) \notin \Omega_S, \\ n_1 \dfrac{\exp\left(\dfrac{d_{\Omega_S} - d(x,y)}{d_{\Omega_S}}\right)^{n_2} - 1}{\exp(1) - 1}, & (x,y) \in \Omega_S, \end{cases} \tag{30.32}$$

where n_1 and n_2 are, in general, experimental parameters, d_{Ω_S} is the sponge-layer diameter and $d(x,y)$ stands for a distance function between a point (x,y) and the intersection of Γ with the boundary of Ω_S (see, e.g., (Walkley, 1999)).

30.6 Numerical Methods

We start this section by noting that a detailed description of the implemented numerical methods referred bellow can be found in the work of N. Lopes (N.Lopes, 2007).

For simplicity, we only consider the second-order system described by equations (30.18) restricted to a stationary bottom and without dissipative, surface tension or extra source terms.

The model variational formulation is written as follows:

$$\left\{\begin{aligned}
&\int_\Omega \frac{\partial\eta}{\partial t}\vartheta_1 \,\mathrm{d}x\mathrm{d}y + \frac{1}{2}\int_\Omega h^2\nabla\left(\frac{\partial\eta}{\partial t}\right)\cdot\nabla\vartheta_1 \,\mathrm{d}x\mathrm{d}y - \frac{1}{6}\int_\Omega \nabla\left(\frac{\partial\eta}{\partial t}\right)\cdot\nabla(h^2\vartheta_1)\,\mathrm{d}x\mathrm{d}y + \\
&\qquad + \frac{1}{15}\int_\Omega h\nabla\left(h\frac{\partial\eta}{\partial t}\right)\cdot\nabla\vartheta_1\,\mathrm{d}x\mathrm{d}y - \frac{1}{15}\int_\Gamma h\frac{\partial h}{\partial\vec{n}}\frac{\partial\eta}{\partial t}\vartheta_1\,d\Gamma = \\
&\int_\Omega (h+\eta)\nabla\Phi\cdot\nabla\vartheta_1\,\mathrm{d}x\mathrm{d}y - \int_\Gamma (h+\eta)\frac{\partial\Phi}{\partial\vec{n}}\vartheta_1\,d\Gamma + \frac{2}{5}\int_\Gamma h^2\frac{\partial}{\partial t}\left(\frac{\partial\eta}{\partial\vec{n}}\right)\vartheta_1 d\Gamma, \\
&\int_\Omega \frac{\partial\Phi}{\partial t}\vartheta_2\,\mathrm{d}x\mathrm{d}y = -\frac{1}{2}\int_\Omega |\nabla\Phi|^2\vartheta_2\,\mathrm{d}x\mathrm{d}y - g\int_\Omega \eta\,\vartheta_2\,\mathrm{d}x\mathrm{d}y - \\
&\qquad - \frac{g}{15}\int_\Omega h\nabla\eta\cdot\nabla(h\vartheta_2)\,\mathrm{d}x\mathrm{d}y + \frac{g}{15}\int_\Gamma h^2\frac{\partial\eta}{\partial\vec{n}}\vartheta_2\,d\Gamma,
\end{aligned}\right. \tag{30.33}$$

where the unknown functions η and Φ are the surface elevation and the transformed velocity potential, whereas ϑ_1 and ϑ_2 are the test functions defined in appropriate spaces.

The spatial discretization of these equations is implemented using low order Lagrange finite elements. In addition, the numerical implementation of (30.33) is accomplished using FFC.

We use a predictor-corrector scheme with an initialization provided by an explicit Runge-Kutta method for the time integration. Note that the discretization of equations (30.33) can be written in the following form:

$$M\dot{U} = \vec{F}(t, U), \tag{30.34}$$

where $\dot{U}$ and U refer to $\left(\frac{\partial\eta}{\partial t}, \frac{\partial\Phi}{\partial t}\right)$ and (η, Φ), respectively. The coefficient matrix M is given by the left-hand sides of (30.33), whereas the known vector $\vec{F}$ is related with the right-hand sides of the same equations. In this way, the fourth order Adams-Bashforth-Moulton method can be written as follows:

$$\left\{\begin{aligned}
MU_{n+1}^{(0)} &= MU_n + \frac{\Delta t}{24}[55\vec{F}(t_n, U_n) - 59\vec{F}(t_{n-1}, U_{n-1}) + \\
&\qquad\qquad + 37\vec{F}(t_{n-2}, U_{n-2}) - 9\vec{F}(t_{n-3}, U_{n-3})], \\
MU_{n+1}^{(1)} &= MU_n + \frac{\Delta t}{24}[9\vec{F}(t_{n+1}, U_{n+1}^{(0)}) + 19\vec{F}(t_n, U_n) - \\
&\qquad\qquad - 5\vec{F}(t_{n-1}, U_{n-1}) + \vec{F}(t_{n-2}, U_{n-2})],
\end{aligned}\right. \tag{30.35}$$

where Δt is the time step, $t_n = n\Delta t$ ($n \in \mathbb{N}$) and U_n is U evaluated at t_n. The predicted and corrected values of U_n are denoted by $U_n^{(0)}$ and $U_n^{(1)}$, respectively. The corrector-step equation ($(30.35)_2$) can be iterated as function of a predefined error between consecutive time steps. For more details see, e.g., (Hairer and Wanner, 1991) or (Lambert, 1991).

30.7 Numerical Applications

In this section, we present some numerical results about the propagation of surface water waves in an harbour with a geometry similar to that one of Fig. 30.1.

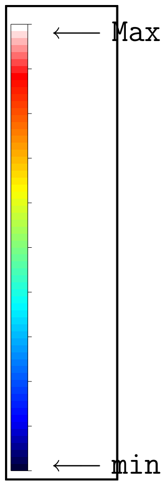
The colour scale used in Figs. 30.5–30.8 is presented in Fig. 30.4. A schematic description of the fluid domain, namely the bottom profile and the sponge layer can be seen in Figs. 30.5 and 30.6, respectively. Note that a piecewise linear bathymetry is considered. A sponge layer is used to absorb the wave energy at the outflow region and to avoid strong interaction between incident and reflected waves in the harbour entrance. A monochromatic periodic wave is introduced at the indicated boundary (Dirichlet BC) in Fig. 30.6. This is achieved by considering waves induced by a periodic Dirichlet boundary condition, as described in the subsection 30.4.2, with the following characteristics:

a	wave amplitude	$0.25\,\mathtt{m}$
ω	wave angular frequency	$0.64715\,\mathtt{s}^{-1}$
p	wave period	$4.06614\,\mathtt{s}$
k	wave number	$0.06185\,\mathtt{m}^{-1}$
L	wave length	$101.59474\,\mathtt{m}$
b	wave potential magnitude	$3.97151\,\mathtt{m}^2\mathtt{s}^{-1}$
c	wave velocity magnitude	$0.24562\,\mathtt{m}\,\mathtt{s}^{-1}$
ε	small amplitude parameter	0.01823
μ	long wave parameter	0.13501

Full reflective walls are assumed as boundary conditions in all domain boundary except in the harbour entrance. In Fig. 30.7 a snapshot of the surface elevation is shown at the time $t_s = 137\,\mathrm{s}$.

A zoom of the image, which describes the physical potential $\phi_0(x, y)$ and velocity vector field in the still water plane, is given in the neighbourhood of the point $P_3 = (255, -75)\,\mathtt{m}$ at t_s (see Fig. 30.8). The Figs. 30.9 and 30.10 represent the surface elevation and water speed as a function of the time, at the points $P_1 = (-350, 150)\,\mathtt{m}$, $P_2 = (-125, 60)\,\mathtt{m}$ and P_3.

From these numerical results, one can conclude that the interaction between incident and reflected waves, near the harbour entrance, can generate wave amplitudes that almost take the triple value of the incident wave amplitude. One can also observe an analogous behaviour for velocities. Note that no mechanism



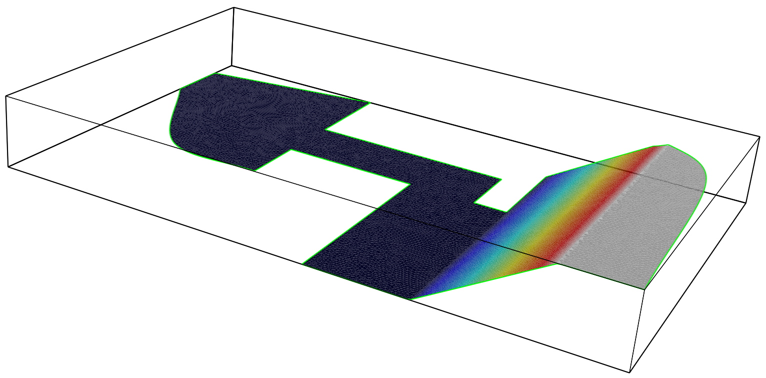


Figure 30.4: Scale. Figure 30.5: Impermeable bottom [$\texttt{Max} = -5.316\,\texttt{m}, \texttt{min} = -13.716\,\texttt{m}$].

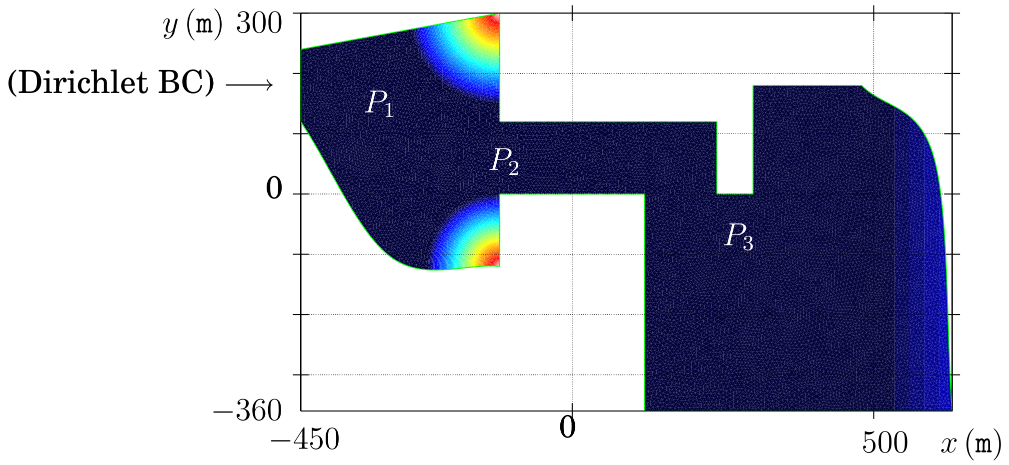


Figure 30.6: Sponge layer (viscosity $\nu(x, y)$) [$\texttt{Max} \approx 0.1\,\texttt{m}^2\texttt{s}^{-1}, \texttt{min} = 0\,\texttt{m}^2\texttt{s}^{-1}$].

for releasing energy of the reflected waves throughout the incident wave boundary is considered.

30.8 Conclusions and future work

As far as we know, the finite element method is not often applied in surface water wave models based on the BEP formulation. In general, finite difference methods are preferred, since they could be easily applied to higher-order equations. On the other hand, they are not appropriated for the treatment of complex geometries, like those of harbours, for instance.

In fact, the surface water wave problems are associated with Boussinesq-type

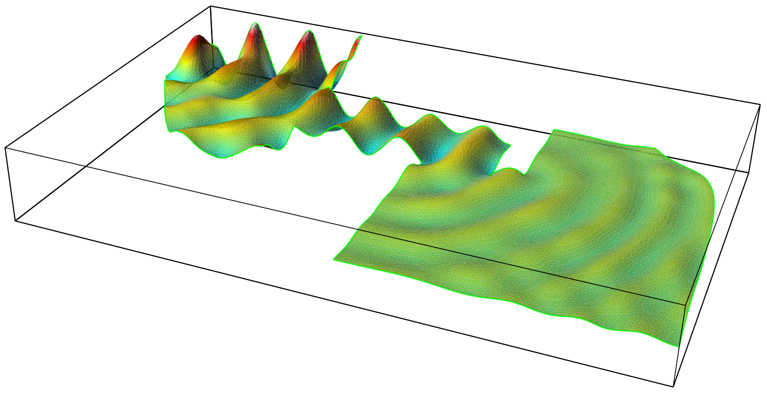


Figure 30.7: Surface elevation [$\texttt{Max} \approx 0.63\,\texttt{m}, \texttt{min} \approx -0.73\,\texttt{m}$].

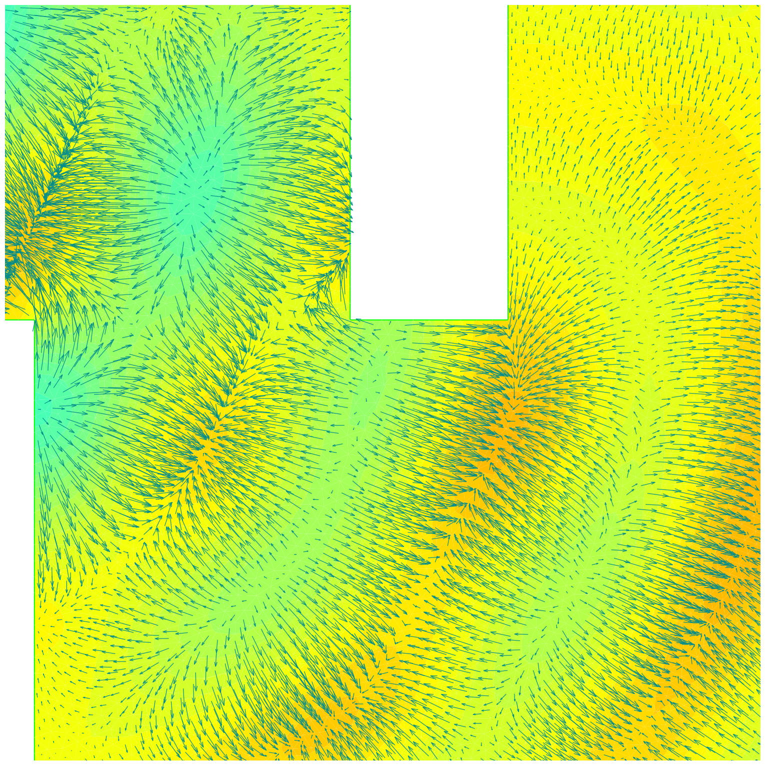


Figure 30.8: Velocity vector field at $z = 0$ and potential $\phi_0(x, y, t_s)$ near P_3. Potential values in Ω: [$\texttt{Max} \approx 14.2\,\texttt{m}^2\texttt{s}^{-1}, \texttt{min} = -12.8\,\texttt{m}^2\texttt{s}^{-1}$].

governing equations, which require very high order (≥ 6) spatial derivatives or a very high number of equations (≥ 6). A first approach, to the high-order models using discontinuous Galerkin finite element methods, can be found in (Engsig-Karup et al., 2006).

From this work one can conclude that the FEniCS packages, namely DOLFIN and FFC, are appropriated to model surface water waves.

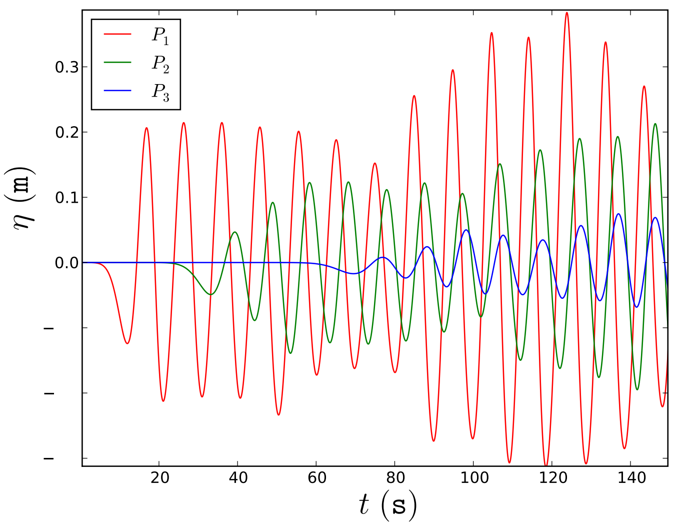


Figure 30.9: Surface elevation at P_1, P_2 and P_3 $[\mathtt{Max} \approx 0.4\,\mathtt{m}, \mathtt{min} = -0.31\,\mathtt{m}]$.

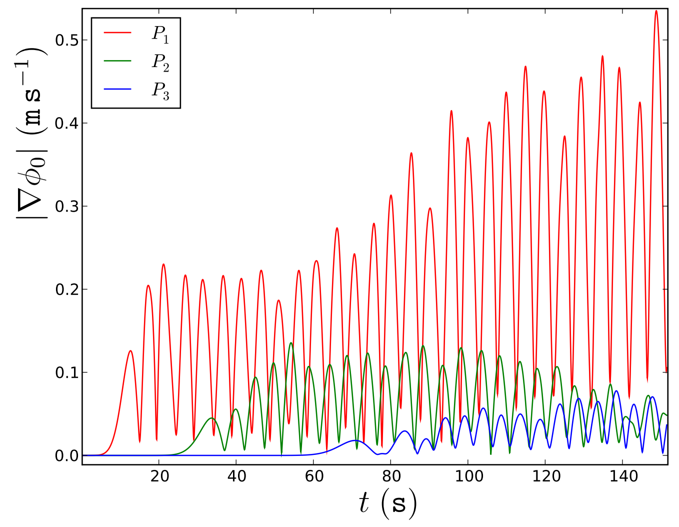


Figure 30.10: Water speed at P_1, P_2 and P_3 $[\mathtt{Max} \approx 0.53\,\mathtt{m\,s}^{-1}, \mathtt{min} = 0\,\mathtt{m\,s}^{-1}]$.

We have been developing DOLFWAVE, i.e., a FEniCS based application for BEP models (see `http://www.fenics.org/wiki/DOLFWAVE`). DOLFWAVE will also be compatible with Xd3d post-processor[3].

The current state of the work, along with several numerical simulations, can be found at `http://ptmat.fc.ul.pt/~ndl`. This package will include some standard potential models of low order (≤ 4) as well as other new models to be

[3]`http://www.cmap.polytechnique.fr/~jouve/xd3d/`

submitted elsewhere by the authors (N.Lopes et al.).

30.9 Acknowledgments

L. Trabucho work is partially supported by Fundação para a Ciência e Tecnologia, Financiamento Base 2008-ISFL-1-209. N. Lopes work is supported by Instituto Superior de Engenharia de Lisboa – ISEL.

N. Lopes[*,◇,†], e-mail: ndl@ptmat.fc.ul.pt
P. Pereira[*,♯],e-mail: ppereira@deq.isel.ipl.pt
L. Trabucho[◇,†],e-mail: trabucho@ptmat.fc.ul.pt

[*] Área Científica de Matemática
ISEL-Instituto Superior de Engenharia de Lisboa,
Rua Conselheiro Emídio Navarro, 1
1959-007 Lisboa
Portugal

[◇] Departamento de Matemática
FCT-UNL-Faculdade de Ciências e Tecnologia
2829-516 Caparica
Portugal

[†] CMAF-Centro de Matemática e Aplicações Fundamentais,
Av. Prof. Gama Pinto, 2
1649-003 Lisboa,
Portugal

[♯] CEFITEC-Centro de Física e Investigação Tecnológica
FCT-UNL-Faculdade de Ciências e Tecnologia
2829-516 Caparica
Portugal

CHAPTER 31

Multiphase Flow Through Porous Media

By Xuming Shan and Garth N. Wells

Chapter ref: **[shan]**

Summarise work on automated modelling for multiphase flow through porous media.

CHAPTER 32

A coupled stochastic and deterministic model of Ca^{2+} dynamics in the dyadic cleft

By Johan Hake

Chapter ref: **[hake]**

32.1 Introduction

From the time we are children, we are told that we should drink milk because it is an important source of calcium (Ca^{2+}), and that Ca^{2+} is vital for a strong bone structure. What we do not hear as frequently, is that Ca^{2+} is one of the most important cellular messengers in the human body (Alberts et al., 2002). In particular, Ca^{2+} controls cell death, neural signaling, secretion of different chemical substances to the body, and the focus of this chapter: the contraction of cells in the heart.

In this chapter, we will present a computational model that can be used to model Ca^{2+} dynamics in a small sub-cellular domain called the dyadic cleft. The model includes Ca^{2+} diffusion, which is described by an advection-diffusion partial differential equation, and discrete channel dynamics, which is described by stochastic Markov models. Numerical methods implemented in PyDOLFINsolving the partial differential equation will also be pregsented. In the last section, we describe a time stepping scheme that is used to solve the stochastic and deterministic models. We will also present a solver framework, DiffSim, that implements the time stepping scheme together with the numerical methods solving the computational model described above.

32.2 Biological background

In a healthy heart, every heart beat originates in the sinusoidal node, where pacemaker cells trigger an electric signal. This signal is a difference in electric potential between the interior and exterior of the heart cells. These two domains are separated by the cell membrane. The difference in the electric potential between these domains is called the membrane potential. The membrane potential propagates through the whole heart using active conductances at the cell membrane. The actively propagating membrane potential is called an action potential. When an action potential arrives at a heart cell, it triggers the L-type Ca^{2+} channels (LCCs). These channels bring Ca^{2+} into the cell. Some of the Ca^{2+} diffuse over a small cleft, called the dyadic cleft, and cause further Ca^{2+} release from an intracellular Ca^{2+} storage, the sarcoplasmic reticulum (SR), through a channel called the ryanodine receptor (RyR). The Ca^{2+} ions then diffuse to the main intracellular domain of the cell, the cytosole, in which the contractile proteins are situated. The Ca^{2+} ions attach to these proteins and trigger contraction. The strength of the contraction is controlled by the strength of the Ca^{2+} concentration ($\left[Ca^{2+}\right]$) in cytosole. The contraction is succeeded by a period of relaxation, which is caused by the extraction of Ca^{2+} from the intracellular space by various proteins.

This chain of events is labelled the Excitation Contraction (EC) coupling (Bers, 2001). Several severe heart diseases can be related to impaired EC coupling. By broadening the knowledge of the coupling, it may be possible to develop better treatments for such diseases. Although the big picture of EC coupling is straightforward to grasp, it involves the nonlinear action of hundreds of different protein species. Computational methods have emerged as a natural complement to experimental studies to better understand the intriguing coupling. In this chapter, we focus on the initial phase of the EC coupling, the stage where Ca^{2+} flows into the cell and triggers further Ca^{2+} release.

32.3 Mathematical models

In this section we describe the computational model for the early phase of the EC coupling. We first present the morphology of the cleft, and how we model this in our study. Then we describe the mathematical equation for the diffusion of Ca^{2+} inside the cleft together with the boundary fluxes. Finally, we discuss the stochastic models that govern the discrete channel dynamics of the LCCs and RyRs.

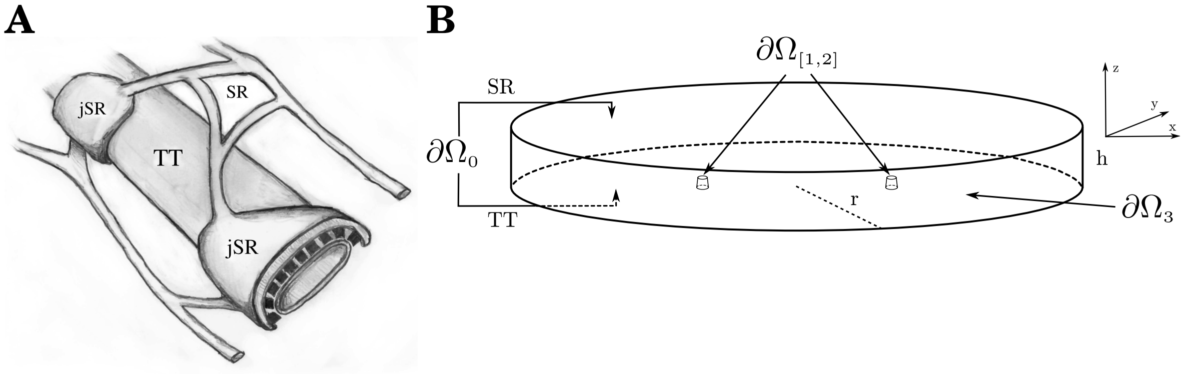


Figure 32.1: **A**: A diagram showing the relationship between the TT, the SR, and the jSR. The volume between the flat jSR and the TT is the dyadic cleft. The black structures in the cleft are Ryanodine receptors, which are large channel proteins. **B**: The geometry used for the dyadic cleft. The top of the disk is the cell membrane of the SR or jSR. The bottom is the cell membrane of the TT, and the circumference of the disk is the interface to the cytosole. The elevations in the TT membrane models two ions channels.

Morphology

The dyadic cleft is the volume between a structure called the t-tubule (TT) and the SR. The TT is a network of pipe-like invaginations of the cell membrane that perforate the heart cell (Soeller and Cannell, 1999). In Fig. 32.1 **A**, a sketch of a small part of a single TT together with a piece of SR is presented. Here we see that the junctional SR (jSR) is wrapped around the TT. The small volume between these two structures is the dyadic cleft. The space is not well defined as it is crowded with channel proteins, and its size also varies. In computational studies, it is commonly approximated as a disk or a rectangular slab (Koh et al., 2006, Peskoff et al., 1992, Soeller and Cannell, 1997, Tanskanen et al., 2007). In this study a disk with height, h = 12 nm and radius, r = 50 nm has been used for the domain Ω, see Fig. 32.1 **B**. The diffusion constant of Ca^{2+} is set to $\sigma = 10^5$ nm^2 ms^{-1} (Langer and Peskoff, 1996).

Ca^{2+} Diffusion

Electro-Diffusion

We will use Fick's second law to model the diffusion of Ca^{2+} in the dyadic cleft. Close to the cell membrane, the ions are affected by an electric potential. The potential is caused by negative charges on the membrane (Langner et al., 1990, McLaughlin et al., 1971). The potential attenuates fast as it is screened by the ions in the intracellular solution. We will describe the electric potential using the Gouy-Chapman method (Grahame, 1947). This theory introduces an advection term to the standard diffusion equation, which makes the resulting equation harder to solve. To simplify the presentation we will use a non-dimensional electric potential ψ, which is the electric potential scaled by a factor of e/kT. Here e is the electron charge, k is Boltzmann's constant and T is the temperature. We will also use a non-dimensional electric field which is given by:

$$E = -\nabla\psi. \tag{32.1}$$

The Ca^{2+} flux in a solution in the presence of an electric field is governed by the Nernst-Planck equation,

$$J = -\sigma\left(\nabla c - 2\,cE\right), \tag{32.2}$$

where $c = c(x,t)$ is the $\left[Ca^{2+}\right]$ ($x \in \Omega$ and $t \in$[0,T]), σ the diffusion constant, $E = E(x)$ the non-dimensional electric field and 2 is the valence of Ca^{2+}. Assuming conservation of mass, we arrive at the advection-diffusion equation,

$$\dot{c} = \sigma\left(\Delta c - \nabla\cdot(2\,cE)\right). \tag{32.3}$$

Here $\dot{c}$ is the time derivative of c.

The strength of ψ is defined by the amount of charge at the cell membrane and by the combined screening effect of all the ions in the dyadic cleft. In addition to Ca^{2+}, the intracellular solution also contains K^+, Na^+, Cl^-, and Mg^{2+}. Following the previous approach by Langner et al. (1990) and Soeller and Cannell (1997), these other ions will be treated as being in steady state. The cell membrane is assumed to be planar and effectively infinite. This assumption allows us to use an approximation of the electric potential in the solution,

$$\psi(z) = \psi_0\,\mathrm{e}^{-\kappa z}. \tag{32.4}$$

Here ψ_0 is the non-dimensional potential at the membrane, κ the inverse Debye length and z the distance from the cell membrane. We will use $\psi_0 = -2.2$ and $\kappa = 1$ nm.

Boundary fluxes

The boundary, $\partial\Omega$, is divided into 4 disjoint boundaries, $\partial\Omega_k$, for $k = 1,\ldots,4$, see Fig. 32.1 **B**. To each boundary we associate a flux, $J_{|\partial\Omega_k} = J_k$. The SR and TT membranes are impermeable for ions, effectively making $\partial\Omega_1$, in Fig. 32.1 **B**, a no-flux boundary, giving us,

$$J_1 = 0. \tag{32.5}$$

We include 2 LCCs in our model. The Ca^{2+} flows into the cleft at the $\partial\Omega_{[2,3]}$ boundaries, see Fig. 32.1 **B**. Ca^{2+} entering these channels then diffuse to the RyRs triggering Ca^{2+} release from the SR. This additional Ca^{2+} flux will not be included

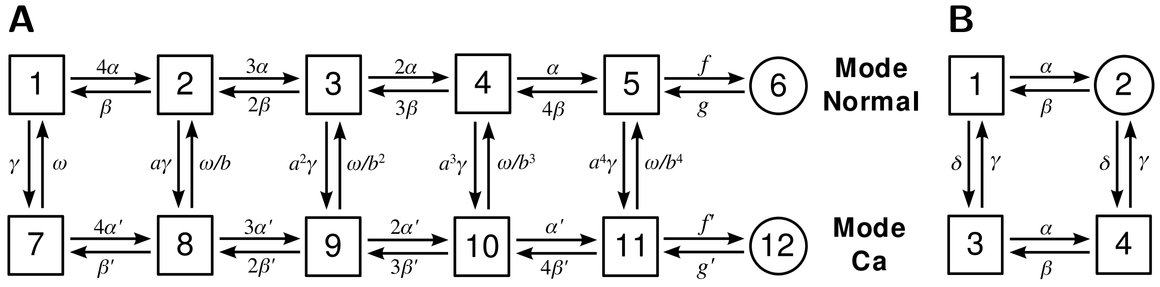


Figure 32.2: **A**: State diagram of the discrete LCC Markov model from Jafri et al. (1998). Each channel can be in one of the 12 states. The transitions between the states are controlled by propensities. The α, and β are voltage dependent, γ is $\left[\mathrm{Ca}^{2+}\right]$ dependent and f, a, b, and ω are constant, see Jafri et al. (1998) for further details. The channels operate in two modes: *Mode normal*, represented by the states in the upper row, and *Mode Ca*, represented the states in the lower row. In state 6 and 12 the channel is open, but state 12 is rarely entered as $f' \ll f$, effectively making *Mode Ca* an inactivated mode. **B**: State diagram of an RyR from Stern et al. (1999). The α and γ propensities are Ca^{2+} dependent, representing the activation and inactivation dependency of the cytosolic $\left[\mathrm{Ca}^{2+}\right]$. The β and δ propensities are constant.

in the simulations. However, the stochastic dynamics of the opening of the channel will be included. Further detailes are presented in Section 32.3 below. The Ca^{2+} that enters the dyadic cleft diffuses into the main compartment of cytosole, introducing a third flux. This flux is included in the model at the $\partial\Omega_3$ boundary.

The LCC is a stochastic channel that takes the state of either open or closed. When the channel is open, Ca^{2+} flows into the cleft. The dynamic that describe the stochastic behaviour is presented in Section 32.3 below. The current amplitude of an open LCC channel is modelled to -0.1 pA (Guia et al., 2001). The LCC flux is then,

$$J_{[2,3]} = \begin{cases} 0 & : \text{ closed channel} \\ -\frac{i}{2FA}, & : \text{ open channel} \end{cases} \qquad (32.6)$$

where i is the amplitude, 2 the valence of Ca^{2+}, F Faraday's constant and A the area of the channel. Note that an inward current is by convention negative.

The flux to the cytosole is modeled as a concentration dependent flux,

$$J_4 = -\sigma\frac{c - c_0}{\Delta s}, \qquad (32.7)$$

where c is the concentration in the cleft at the boundary, c_0 the concentration in the cytosole, and $\Delta s$is an approximation of the distance to the center of the cytosole. In our model we have used Δs= 50 nm.

Stochastic models of single channels

Discrete and stochastic Markov chain models are used to describe single channel dynamics. Such models are described by a certain number of discrete states. Each channel can be in either one of these states. A transition between two states is a stochastic event. The frequency of these events are determined by the propensity functions associated with each transition. These functions, which may vary with time, characterize the probability per unit time that the corresponding transition event occurs. Each Markov model defines its own propensity functions.

L-type Ca^{2+} channel

The LCC opens when an action potential arrives at the cell. The channel inactivates when single Ca^{2+} ions bind to binding sites on the intracellular side of the channel. An LCC is composed of a complex of four transmembrane subunits. Each of these can be permissive or non-permissive. For the whole channel to be open, all four subunits need to be permissive and the channel then has to undergo a last conformational change to an opened state (Hille, 2001). In this chapter we are going to use a Markov model of the LCC that incorporates a voltage dependent activation together with a Ca^{2+} dependent inactivation (Jafri et al., 1998, **?**). The state diagram of this model is presented in Fig. 32.2 **A**. It consists of 12 states, where state 6 and 12 are the only conducting states, hence defineing the open states. The transition propensities are defined by a set of functions and constants, which are all described in Greenstein and Winslow (2002).

Ryanodine Receptors

RyRs are Ca^{2+} specific channels that are gathered in clusters at the SR membrane in the dyadic cleft. These clusters can consist of several hundreds of RyRs (Beuckelmann and Wier, 1988, Franzini-Armstrong et al., 1999). They open by single Ca^{2+} ions attaching to the receptors at the cytosolic side. We will use a modified version of a phenomenological RyR model that mimics the physiological functions of the channel (Stern et al., 1999). The model consists of four states where only one is conducting, state 2, see Fig. 32.2 **B**. The α and γ propensities are Ca^{2+} dependent, representing the activation and inactivation dependency of cytosolic $\left[Ca^{2+}\right]$. The β and δ propensities are constants. For specific values for the propensities, see Stern et al. (1999).

```
from numpy import *
from dolfin import *

mesh = Mesh('cleft_mesh.xml.gz')

Vs = FunctionSpace(mesh, "CG", 1)
Vv = VectorFunctionSpace(mesh, "CG", 1)

v = TestFunction(Vs)
u = TrialFunction(Vs)

# Defining the electric field-function
a = Expression(["0.0","0.0","phi_0*valence*kappa*sigma*exp(-kappa*x[2])"],
               defaults = {"phi_0":-2.2,"valence":2,"kappa":1,"sigma":1.e5},
               V = Vv)

# Assembly of the K, M and A matrices
K = assemble(inner(grad(u),grad(v))*dx)
M = assemble(u*v*dx)
E = assemble(-u*inner(a,grad(v))*dx)

# Collecting face markers from a file, and skip the 0 one
sub_domains = MeshFunction("uint",mesh,"cleft_mesh_face_markers.xml.gz")
unique_sub_domains = list(set(sub_domains.values()))
unique_sub_domains.remove(0)

# Assemble matrices and source vectors from exterior facets domains
domain = MeshFunction("uint",mesh,2)
F = {};f = {};tmp = K.copy(); tmp.zero()
for k in unique_sub_domains:
    domain.values()[:] = (sub_domains.values() != k)
    F[k] = assemble(u*v*ds, exterior_facet_domains = domain, \
                    tensor = tmp.copy(), reset_sparsity = False)
    f[k] = assemble(v*ds, exterior_facet_domains = domain)
```

Figure 32.3: Python code for the assembly of the matrices and vectors from Eq. (32.14)-(32.15).

32.4 Numerical methods for the continuous system

In this section, we will describe the numerical methods used to solve the continuous part of the computational model of the Ca^{2+} dynamics in the dyadic cleft. We will provide PyDOLFINcode for each part of the presentation. The first part of the section describes the discretization of the continuous problem using a finite element method. The second part describes a method to stabilize the discretization. In this part, we also conduct a parameter study to find the optimal stabilization parameters.

Discretization

The continuous problem is defined by Eqs. (32.3 -32.7) together with an initial condition. Given a bounded domain $\Omega \subset \mathbb{R}^3$ with the boundary, $\partial\Omega$, we want to find $c = c(x,t) \in \mathbb{R}_+$, for $x \in \Omega$ and $t \in [0,T]$, such that:

$$\left\{ \begin{array}{rcll} \dot{c} & = & \sigma\Delta c - \nabla \cdot (ca) & \text{in } \Omega \\ \sigma\partial_n c - ca\cdot n & = & J_k & \text{on } \partial\Omega_k,\ k = 1,\dots,4, \end{array} \right. \tag{32.8}$$

and $c(\cdot,0) = c_0(x)$. Here $a = a(x) = 2\sigma E(x)$ and J_k is the k^{th} flux at the k^{th} boundary $\partial\Omega_k$, where $\bigcup_{k=1}^4 \partial\Omega_k = \partial\Omega$, $\partial_n c = \nabla c \cdot n$, where n is the outward normal on the boundary. The J_k are given by Eqs. (32.5)- (32.7).

The continuous equations are discretized using a finite element method in space. Eq. (32.8) is multiplied with a proper test function v, and integrated over the spatial domain, thus obtaining:

$$\int_\Omega \dot{c}v\,dx = \int_\Omega (\sigma\Delta c - \nabla(ca))\,v\,dx. \tag{32.9}$$

Integration by parts, together with the boundary conditions in Eq. (32.8), yield:

$$\int_\Omega \dot{c}v\,dx = -\int_\Omega (\sigma\nabla c - ca)\cdot\nabla v\,dx + \sum_k \int_{\partial\Omega_k} J_k v\,ds_k. \tag{32.10}$$

Consider a mesh $\mathcal{T} = \{K\}$ of simplicial elements K. Let V_h denote the space of piecewise linear polynomials, defined relative to the mesh $\mathcal{T}$. Using the backward Euler methods in time, we seek an approximation of c: $c_h \in V_h$ with nodal basis $\{\phi_i\}_{i=1}^N$. Eq. (32.10) can now be discretized as follows: Consider the n^{th} time step, then given c_h^n find $c_h^{n+1} \in V_h$ such that

$$\int_\Omega \frac{c_h^{n+1} - c_h^n}{\Delta t} v\,dx = -\int_\Omega \left(\sigma\nabla c_h^{n+1} - c_h^{n+1}a\right)\cdot\nabla v\,dx + \sum_k \int_{\partial\Omega} J_k v\,ds_k \quad \forall v \in V_h, \tag{32.11}$$

where $\Delta t$is the size of the time step. The trial function $c_h^n(x)$ is expressed as a weighted sum of basis functions,

$$c_h^n(x) = \sum_j^N C_j^n \phi_j(x). \tag{32.12}$$

where C_j^n are the coefficients. Due to the choice of V_h will the number of unknowns, N, coincide with the number of vertices of the mesh.

Taking test functions, $v = \phi_i$, $i \in \{1,\dots,N\}$, gives the following algebraic system of equations in terms of the coefficients $\left\{c_i^{n+1}\right\}_{i=1}^N$.

$$\frac{1}{\Delta t}\boldsymbol{M}\left(C^{n+1} - C^n\right) = \left(-\boldsymbol{K} + \boldsymbol{E} + \sum_k \alpha^k \boldsymbol{F}^k\right) C_j^{n+1} + \sum_k c_0^k\, f^k. \tag{32.13}$$

Here $C^n \in \mathbb{R}^N$ is the vector of coefficients from the discrete solution $c_h^n(x)$, α^k and c_0^k are constant coefficients given by Eqs. (32.5) - (32.7) and

```
# Defining the stabilization using local Peclet number
cppcode = """class Stab : public Expression {
public:
  Expression* field; double sigma;
  Stab(): Expression(3), field(0), sigma(1.0e5)
  {
    value_shape.push_back(3);
  }
  void eval(double* v, const Data& data) const {
    if (!field)
      error("Attach a field function.");
    double field_norm = 0.0; double tau = 0.0;
    double h = data.cell().diameter();
    field->eval(v,data);
    for (uint i = 0;i < geometric_dimension(); ++i)
      field_norm += v[i]*v[i];
    field_norm = sqrt(field_norm);
    double PE = 0.5*field_norm * h/sigma;
    if (PE > DOLFIN_EPS)
      tau = 1/tanh(PE)-1/PE;
    for (uint i = 0;i < geometric_dimension(); ++i)
      v[i] *= 0.5*h*tau/field_norm;}};
"""
stab = Expression(cppcode, V = Vv); stab.field = a

# Assemble the stabilization matrices
E_stab = assemble(div(a*u)*inner(stab,grad(v))*dx)
M_stab = assemble(u*inner(stab,grad(v))*dx)

# Adding them to the A and M matrices, weighted by the global tau
tau = 0.28; E.axpy(tau,E_stab,True); M.axpy(tau,M_stab,True)
```

Figure 32.4: Python code for the assembly of the SUPG term for the mass and advection matrices.

$$\begin{aligned} \boldsymbol{M}_{ij} &= \int_\Omega \phi_i \phi_j dx, & \boldsymbol{K}_{ij} &= \int_\Omega \nabla \phi_i \cdot \nabla \phi_j dx, \\ \boldsymbol{E}_{ij} &= \int_\Omega a\phi_i \cdot \nabla \phi_j dx, & \boldsymbol{F}^k_{ij} &= \int_{\partial\Omega_k} \phi_i \phi_j ds, \end{aligned} \tag{32.14}$$

are the entries in the $\boldsymbol{M}$, $\boldsymbol{K}$, $\boldsymbol{E}$ and $\boldsymbol{F}^k$ matrices. f^k are boundary source vectors corresponding to the k^{th} boundary. The vector elements are given by:

$$f^k_i = \int_{\partial\Omega_k} \phi_i ds. \tag{32.15}$$

The PyDOLFINcode for the assembly of the matrices and vectors in Eqs. (32.14)-(32.15) is presented in Fig. 32.3. Note that we define only one form for the different boundary mass matrices and boundary source vectors, `u*v*ds` and `v*ds` respectively. The `assemble` routine will assemble these forms over the 0^{th} sub-domain, see line 31 and 33 in Fig. 32.3. By passing sub domain specific `MeshFunctions` to the assemble routine we can assemble the correct boundary mass matrices and boundary source vectors. We collect the matrices and boundary source vectors. These are then added to form the linear system to be solved at each time step. If an LCC opens, we get contributions to the right-hand side from the source vectors. If an LCC closes, the same source vectors are removed from the right-hand side. When an LCC either opens or closes, a large flux is either added or removed from the system. To be able to resolve the sharp time gradients correctly, we need to take smaller time steps after such an event. The time step is then expanded by multiply it with a constant ¿ 1.

The sparse linear system is solved using the PETSclinear algebra backend (Balay et al., 2001) in PyDOLFINtogether with the Bi-CGSTAB iterative solver (van der Vorst, 1992), and the BoomerAMG preconditioners from hypre (Falgout and Yang, 2002). In Fig. 32.5, a script is presented that solves the algebraic system from Eq. (32.13) together with a crude time stepping scheme for the opening and closing of the included LCC channel.

Stabilization

It turns out that the algebraic system in Eq. (32.13) can be numerically unstable for physiological relevant values of a. This is due to the transport term introduced by $\boldsymbol{E}_{ij}$ from Eq. (32.14). We have chosen to stabilize the system using the Streamline upwind Petrov-Galerkin (SUPG) method (Brooks and Hughes, 1982). This method adds an upwind discontinuous contribution to the testfunction in the streamline direction Eq. (32.9),

$$v' = v + s, \text{ where } s = \tau \frac{h\tau_e}{2\|a\|} a \cdot \nabla v. \tag{32.16}$$

Here τ is a parameter we want to optimize (see later in this Section), $\|\cdot\|$ is the Euclidian norm in $\mathbb{R}^3$, $h = h(x)$ is the element size, and $\tau_e = \tau_e(x)$, is given by,

$$\tau_e = \coth(\mathrm{PE_e}) - \frac{1}{\mathrm{PE_e}}, \tag{32.17}$$

where $\mathrm{PE_e}$ is the element Péclet number:

$$\mathrm{PE_e} = \frac{\|a\|h}{2\sigma}. \tag{32.18}$$

When $\mathrm{PE_e}$ is larger than 1 the system become unstable, and oscillations is introduced.

In the 1D case, with a uniform mesh, the stabilization term defined by Eqs. (32.17) - (32.18), can give nodal exact solutions (Brooks and Hughes, 1982, Christie et al., 1976). Our choice of stabilization parameter is inpired by this. We have used the diameter of the sphere that circumscribes the local tetrahedron as h. This is what DOLFINimplements in the function `Cell.diameter()`. We recognize that other choices exist, which might give better stabilization (John and Knobloch, 2007). Tezduyar and Park (1986) use a length based on the size of the element in the direction of a.

The PyDOLFINcode that assembles the SUPG part of the problem is presented in Fig. 32.4. In the script, two matrices, `E_stab` and `M_stab` are assembled. Both matrices are added to the corresponding advection and mass matrices `E` and `M`, weighted by the global parameter `tau`.

A mesh with finer resolution close to the TT surface, at $z = 0$ nm, is used to resolve the steep gradient of the solution in this area. It is here the electric field is at its strongest, yielding an element Péclet number larger than 1. However the field attenuate fast: at $z = 3$ nm the field is down to 5% of the maximal amplitude, and at $z = 5$ nm, it is down to 0.7%.The mesh can thus be fairly coarse in the interior of the domain. The mesh generator `tetgen` is used to to produce meshes with the required resolution (Si, 2007).

The global stabilization parameter τ, is problem dependent. To find an optimal τ, for a certain electric field and mesh, the system in Eq. (32.13) is solved to steady state, defined as T = 1.0 ms, using only homogeneous Neumann boundary conditions. An homogeneous concentration of $c_0 = 0.1\ \mu$M is used as the initial condition. The numerical solution is then compared with the analytic solution of the problem. This solution is acquired by setting $J = 0$ in Eq. (32.2) and solving for the c, with the following result:

$$c(z) = c_b\,\mathrm{e}^{-2\psi(z)}\,. \tag{32.19}$$

Here ψ is given by Eq. (32.4), and c_b is the concentration in the bulk, i.e., where z is large. c_b was chosen such that the integral of the analytic solution was equal to $c_0 \times V$, where V is the volume of the domain.

```
# Model parameters
dt_min = 1.0e-10; dt = dt_min; t = 0; c0 = 0.1; tstop = 1.0
events = [0.2,tstop/2,tstop,tstop]; dt_expand = 2.0;
sigma = 1e5; ds = 50; area = pi; Faraday = 0.0965; amp = -0.1
t_channels = {1:[0.2,tstop/2], 2:[tstop/2,tstop]}

# Initialize the solution Function and the left and right hand side
u = Function(Vs); x = u.vector()
x[:] = c0#*exp(-a.valence*a.phi_0*exp(-a.kappa*mesh.coordinates()[:,-1]))
b = Vector(len(x)); A = K.copy();

solver = KrylovSolver("bicgstab","amg_hypre")
solver.parameters["relative_tolerance"] = 1e-10
solver.parameters["absolute_tolerance"] = 1e-7

plot(u, vmin=0, vmax=4000, interactive=True)
while t < tstop:
    # Initalize the left and right hand side
    A.assign(K); A *= sigma; A += E; b[:] = 0

    # Adding channel fluxes
    for c in [1,2]:
        if t >= t_channels[c][0] and t < t_channels[c][1]:
            b.axpy(-amp*1e9/(2*Faraday*area),f[c])

    # Adding cytosole flux at Omega 3
    A.axpy(sigma/ds,F[3],True); b.axpy(c0*sigma/ds,f[3])

    # Applying the Backward Euler time discretization
    A *= dt; b *= dt; b += M*x; A += M

    solver.solve(A,x,b)
    t += dt; print "Ca Concentration solved for t:",t

    # Handle the next time step
    if t == events[0]:
        dt = dt_min; events.pop(0)
    elif t + dt*dt_expand > events[0]:
        dt = events[0] - t
    else:
        dt *= dt_expand

    plot(u, vmin=0, vmax=4000)

plot(u, vmin=0, vmax=4000, interactive=True)
```

Figure 32.5: Python code for solving the system in Eq. (32.13), using the assembled matrices from the two former code examples from Fig. 32.3- 32.4.

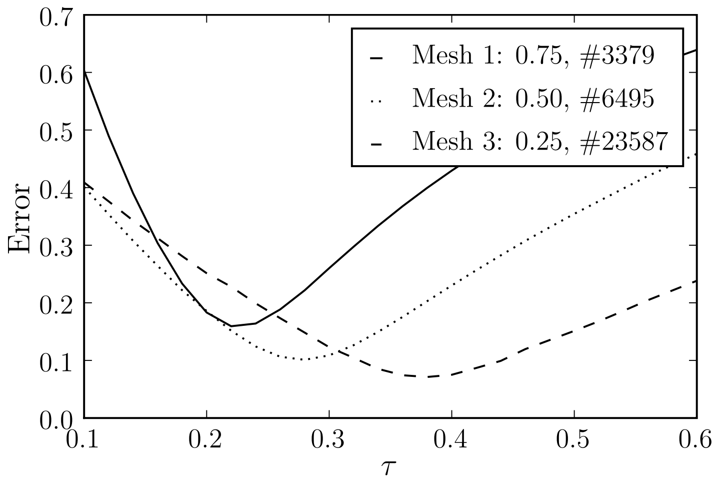


Figure 32.6: The figure shows a plot of the error versus the stabilization parameter τ for 3 different mesh resolutions. The mesh resolutions are given by the median of the z distance of all vertices and the total number of vertices in the mesh, see legend. We see that the minimal values of the error for the three meshes, occur at three different τ: 0.22, 0.28, and 0.38.

The error of the numerical solution for different values of τ and for three different mesh resolutions are plotted in Fig. 32.6. The meshes are enumerated from 1-3. The error is computed using the $L^2(\Omega)$ norm and is normalized by the $L^2(\Omega)$ norm of the analytical solution,

$$\frac{\|c(T) - c_h^{n_T}\|_{L^2}}{\|c(T)\|_{L^2}}, \tag{32.20}$$

where n_T is the time step at $t = T$. As expected, we see that the mesh with the finest resolution produces the smallest error. The mesh resolutions are quantified by the number of vertices close to $z = 0$. In the legend of Fig. 32.6, the median of the z distance of all vertices and the total number of vertices in each mesh is presented. The three meshes were created such that the vertices closed to $z = 0$ were forced to be situated at some fixed distances from $z = 0$. Three numerical and one analytical solution for the three different meshes are plotted in Fig. 32.7- 32.9. The numerical solutions are from simulations using three different τ: 0.1, 0.6 and the L^2-optimal τ, see Fig. 32.6. The traces in the figures are from the discrete solution $c_h^{n_T}$, evaluated on the straight line between the spatial points p_0=(0,0,0) and p_1=(0,0,12).

In Fig. 32.7 the traces from mesh 1 is plotted. Here we see that the numerical solutions are quite poor for some of the τ. The solution with $\tau = 0.10$ is obviously not correct, as it produces negative concentrations. The solution with $\tau = 0.60$ seems more correct but it undershoots the analytic solution at $z = 0$ with $\tilde{1}.7$ μM. The solution with $\tau = 0.22$ is the L^2-optimal solution for mesh 1, and approximates the analytic solution at $z = 0$ well.

In Fig. 32.8 the traces from mesh 2 is presented in two plots. The left plot shows the traces for $z < 1.5$ nm, and the right shows the traces for $z > 10.5$ nm. In the left plot we see the same tendency as in Fig. 32.7, an overshoot of the solution with $\tau = 0.10$ and an undershoot of the solution with $\tau = 0.60$. The L^2-optimal solution, the one with $\tau = 0.28$, overshoots the analytic solution for the shown interval in the left plot, but undershoots for the rest of the trace.

In the last figure, Fig. 32.9, traces from mesh 3 is presented. The results is also here presented in two plots, corresponding to the same z interval as in Fig. 32.8. We see that the solution with $\tau = 0.10$ is not good in either plots. In the left plot it clearly overshoots the analytic solution for most of the interval, and then stays at a lower level than the analytic solution for the rest of the interval. The solution with $\tau = 0.60$ is much better here than in the two previous plots. It undershoots the analytic solution at $z = 0$ but stays closer to it for the rest of the interval than the L^2-optimal solution. The L^2 norm penalize larger distances between two traces, i.e., weighting the error close to $z = 0$ more than the rest. The optimal solution measured in the Max norm is given when $\tau = 50$, result not shown.

These results tell us that it is difficult to obtain accurate numerical solution for the advection-diffusion problem presented in Eq. (32.8). Using a finer mesh close to $z = 0$ could help, but it will create a larger algebraic system. It is interesting to notice that the L^2 optimal solutions is better close to $z = 0$, than other solutions and the solution for the largest τ is better than other for z ¿ 2 nm. For a modeller, these constraints are important to know about; the solution at $z = 0$ and $z = 12$ nm are the most important, as Ca^{2+} interact with other proteins at these points.

32.5 `diffsim` an event driven simulator

In the scripts in Fig. 32.3- 32.5, we show how a simple continuous solver can be built with PyDOLFIN. By preassembling the matrices from Eq. (32.14) a flexible system for adding and removing boundary fluxes corresponding to the state of the channels can be constructed. The script in Fig.32.5 uses fixed time points for the channel state transitions. At these time points we minimize Δt, so we can resolve the sharp time gradient. In between the channel transitions we expand Δt. This simplistic time stepping scheme is sufficient to solve the presented example. However it would be difficult to expand it to also incorporate the time stepping involved with the solution of stochastic Markov models, and other discrete variables. For such scenarios, an event driven simulator called `diffsim` has been developed. In the last subsections in this chapter, the algorithm underlying the time stepping scheme in `diffsim` will be presented. An example of how one can use `diffsim` to describe and solve a model of the Ca^{2+} dynamics in the dyadic cleft will also be demonstrated. The `diffsim` software can freely be downloaded from URL:`http://www.fenics.org/wiki/FEniCS_Apps`.

Stochastic system

The stochastic evolution of the Markov chain models presented in Section 32.3 is determined by a modified Gillespie method (Gillespie, 1977), which resembles the one presented in Rüdiger et al. (2007). Here we will not go into detail of the actual method, but rather explain the part of the method that has importance for the overall time stepping algorithm.

The solution of the included stochastic Markov chain models is stored in a state vector, `S`. Each element in `S` corresponds to one Markov model and the value reflects which state each model is in. The transitions between these states are modelled stochastically and are computed using the modified Gillespie method. This method basically gives us which of the states in `S` changes to what state and when. It is not all such state transitions that are relevant for the continuous system. A transition between two closed states in the LCC model will not have any impact on the boundary fluxes, and can be ignored. Only transitions that either open or close a channel (channel transitions), will be recognized. The modified Gillespie method assumes that any continuous variables that a certain propensity function depends on, are constant during a time step. The error of this assumption is reduced by taking smaller time steps right after a channel transition, as the continuous field is indeed changing dramatically during this time period.

Time stepping algorithm

To simplify the presentation of the time stepping algorithm we only consider one continuous variable, this could for example be the Ca^{2+} field. The framework presented here can be expanded to also handle several continuous variables. We define a base class called `DiscreteObject`, which defines the interface for all discrete objects. A key function of a discrete object is to know when its *next event* is due at. The `DiscreteObject` that has the smallest next event time, gets to define the size of the next Δt. In Python this is easily done by making the `DiscreteObjects` sortable with respect to their next event time. All `DiscreteObjects` is then collected in a list, `discrete_objects` see Fig. 32.10. The `DiscreteObject` with the smallest next event time is then just `min(discrete_objects)`.

An event from a `DiscreteObject` that does not have an impact on the continuous solution will be ignored for example a Markov chain model transition that is not a channel transition. A transition needs to be realized before we can tell if it is a channel transition or not. This is done by *stepping* the `DiscreteObject`, i.e., calling the object's `step()` method. If the method returns `False`, it will not affect the Ca^{2+} field, and we enter the while loop, and a new `DiscreteObject` is picked, see Fig. 32.10. If the object returns `True` when stepped, we exit the loop and continue. Next, we have to update the other discrete objects with the chosen Δt, solve for the Ca^{2+} field, broadcast the solution and last but not least execute the discrete event that is scheduled to happen at Δt.

In Fig. 32.11 we show an example of a possible realization of this algorithm. The example starts at t=2ms at the top-most timeline represented by **A**, and it includes three different types of `DiscreteObjects`: i) `DtExpander`, ii) `StochasticHandler`, and iii) `TStop`. See the figure legend for more details.

`diffsim`: an example

`diffsim` is a versatile event driven simulator that incorporates the time stepping algorithm presented in the previous section together with the infrastructure to solve models with one or more diffusional domains, defined by a computational mesh. Each such domain can have several diffusive ligands. Custom fluxes can easily be included through the framework. The sub module `dyadiccleft` implements some published Markov models that can be used to simulate the stochastic behaviour of a dyad and some convenient boundary fluxes. It also implements the field flux from the lipid bi-layer discussed in Section 32.3. In Fig. 32.12 a runnable script is presented, which simulates the time to release, also called the latency, for a dyad. The two Markov models that is presented in Section 32.3 are here used to model the stochastic dynamics of the RyRs and the LCCs. The simulation is driven by a so called dynamic voltage clamp. The data that defining the voltage clamp is read from a file using utilities from the NumPyPython packages.

32.6 Discussion

We have presented a computational model of the Ca^{2+} dynamics of the dyadic cleft in heart cells. It consists of a coupled stochastic and continuous system. We have showed how one can use PyDOLFINto discretise and solve the continuous system using a finite element method. The continuous system is an advection-diffusion equation that produce unstable discretizations. We investigate how one can use the streamline upwind/Petrov-Galerkin method to stabilize the discretized system. We use three different meshes and find an L^2-optimal global stabilization parameters τ for each mesh.

We do not present a solver for the stochastic system. However we outline a time stepping scheme that can be used to couple the stochastic solver with the presented solver for the continuous system. A simulator DiffSimis briefly introduced, which implements the presented time stepping scheme together with the presented solver for the continuous system.

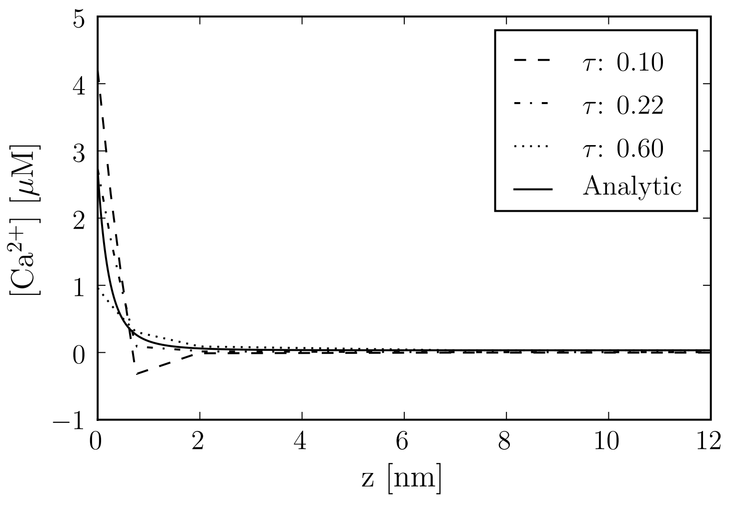


Figure 32.7: The figure shows the concentration traces of the numerical solutions from Mesh 1, see legend of Fig. 32.6, for three different τ together with the analytic solution. The solutions were picked from a line going between the points (0,0,0) and (0,0,12). We see that the solution with $\tau = 0.10$ oscillates. The solution with $\tau = 0.22$ was the solution with smallest global error for this mesh, see Fig 32.6, and the solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0$nm with $\tilde{1}.7$ μM.

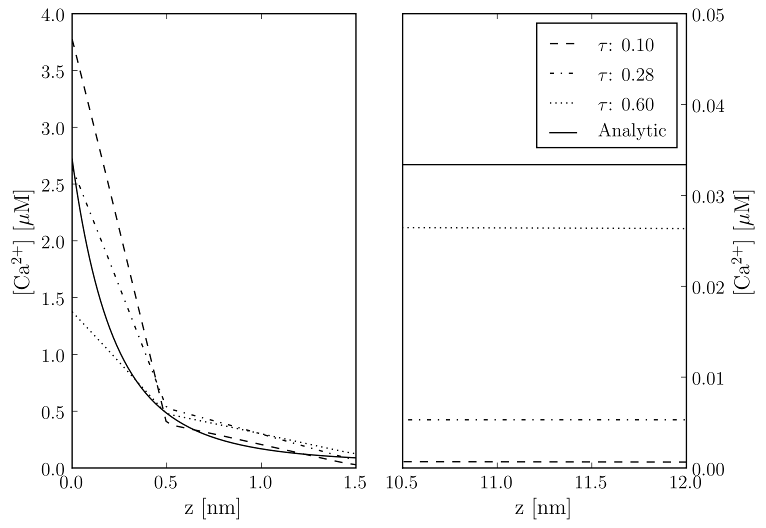


Figure 32.8: The figures show the concentration traces of the numerical solutions from Mesh 2, see legend of Fig. 32.6, for three different τ together with the analytic solution. The solution traces in the two panels are picked from a line going between the spatial points (0,0,0) and (0,0,1.5), for the left panel, and between spatial points (0,0,10.5) and (0,0,12), for the right panel. We see from both panels that the solution with $\tau = 0.10$ give the poorest solution. The solution with $\tau = 0.28$ was the solution with smallest global error for this mesh, see Fig 32.6, and this is reflected in the reasonable good fit seen in the left panel, especially at $z = 0$nm. The solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0$ with $\tilde{1}.2$ μM. From the right panel we see that all numerical solutions undershoot at $z = 15$nm, and that the trace with $\tau = 0.60$ comes the closest to the analytic solution.

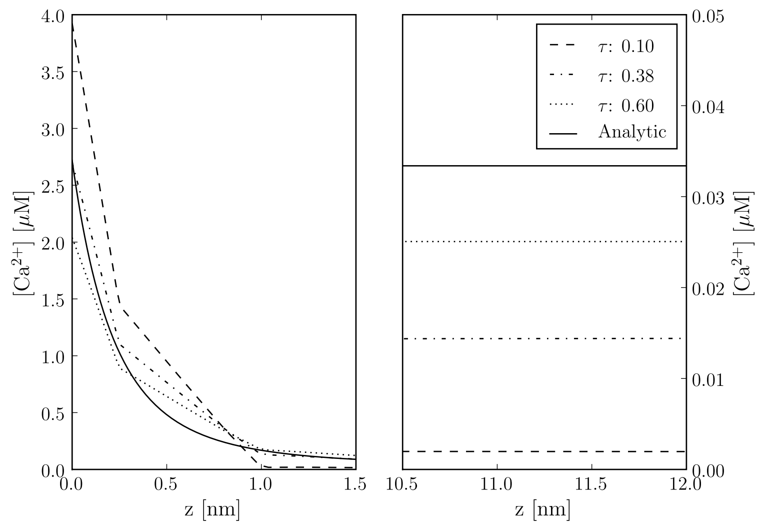


Figure 32.9: The figures shows the concentration traces of the numerical solutions from Mesh 3, see legend of Fig. 32.6, for three different τ together with the analytic solution. The traces in the two panels were picked from the same lines as the one in Fig. 32.8. Again we see from both panels that the solution with $\tau = 0.10$ give the poorest solution. The solution with $\tau = 0.38$ was the solution with smallest global error for this mesh, see Fig 32.6, and this is reflected in the good fit seen in the left panel, especially at $z = 0$nm. The solution with $\tau = 0.60$ undershoots the analytic solution at $z = 0$ with $\tilde{0}.7$ μM. From the right panel we see that all numerical solutions undershoot at $z = 15$nm, and the trace with $\tau = 0.60$ also here comes closest the analytic solution.

```
while not stop_sim:
    # The next event
    event = min(discrete_objects)
    dt = event.next_time()

    # Step the event and check result
    while not event.step():
        event = min(discrete_objects)
        dt = event.next_time()

    # Update the other discrete objects with dt
    for obj in discrete_objects:
        obj.update_time(dt)

    # Solve the continuous equation
    ca_field.solve(dt)
    ca_field.send()

    # Distribute the event
    event.send()
```

Figure 32.10: Python-like pseudo code for the time stepping algorithm used in our simulator

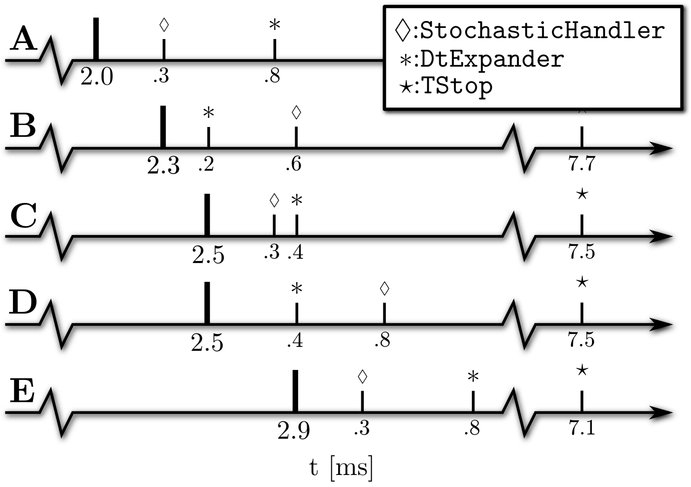


Figure 32.11: Diagram for the time stepping algorithm using 3 discrete objects: `DtExpander`, `StochasticHandler`, `TStop`. The values below the small ticks, corresponds to the time to the next event for each of the discrete objects. This time is measured from the last realized event, which is denoted by the thicker tick. In **A** we have realized a time event at t=2.0 ms. The next event to be realized is a stochastic transition, the one with smallest value below the ticks. In **B** this event is realized, and the `StochasticHandler` now show a new next event time. The event is a channel transition forcing the dt, controlled by the `DtExpander`, to be minimized. `DtExpander` now has the smallest next event time, and is realized in **C**. The channel transition that was realised in **B** raised the $\left[Ca^{2+}\right]$ in the cleft which in turn increase the Ca^{2+} dependent propensity functions in the included Markov models. The time to next event time of the `StochasticHandler` has therefore been updated, and moved forward in **C**. Also note that the `DtExpander` has expanded its next event time. In **D** the stochastic transition is realized and updated with a new next event time, but it is ignored as it is not a channel transition. The smallest time step is now the `DtExpander`, and this is realized in **E**. In this example we do not realize the `TStop` event as it is too far away.

```
from diffsim import *
from diffsim.dyadiccleft import *
from numpy import exp, fromfile

# Model parameters
c0_bulk = 0.1; D_Ca = 1.e5; Ds_cyt = 50; phi0 = -2.2; tau = 0.28
AP_offset = 0.1; dV = 0.5, ryr_scale = 100; end_sim_when_opend = True

# Setting boundary markers
LCC_markers = range(10,14); RyR_markers = range(100,104); Cyt_marker = 3

# Add a diffusion domain
domain = DiffusionDomain("Dyadic_cleft","cleft_mesh_with_RyR.xml.gz")
c0_vec = c0_bulk*exp(-VALENCE[Ca]*phi0*exp(-domain.mesh().coordinates()[:,-1]))

# Add the ligand with fluxes
ligand = DiffusiveLigand(domain.name(),Ca,c0_vec,D_Ca)
field  = StaticField("Bi_lipid_field",domain.name())
Ca_cyt = CytosolicStaticFieldFlux(field,Ca,Cyt_marker,c0_bulk,Ds_cyt)

# Adding channels with Markov models
for m in LCC_markers:
    LCCVoltageDepFlux(domain.name(), m, activator=LCCMarkovModel_Greenstein)
for m in RyR_markers:
    RyRMarkovModel_Stern("RyR_%d"%m, m, end_sim_when_opend)

# Adding a dynamic voltage clamp that drives the LCC Markov model
AP_time = fromfile('AP_time_steps.txt',sep='\n')
dvc = DynamicVoltageClamp(AP_time,fromfile('AP.txt',sep='\n'),AP_offset,dV)

# Get and set parameters
params = get_params()

params.io.save_data = True
params.Bi_lipid_field.tau = tau
params.time.tstop = AP_time[-1] + AP_offset
params.RyRMarkovChain_Stern.scale = ryr_scale

info(str(params))

# Run 10  simulations
data = run_sim(10,"Dyadic_cleft_with_4_RyR_scale")
mean_release_latency = mean([ run["tstop"] for run in data["time"]])
```

Figure 32.12: An example of how `diffsim` can be used to simulate the time to RyR release latency, from a small dyad who's domain is defined by the mesh in the file `cleft_mesh_with_RyR.xml.gz`.

CHAPTER 33

Electromagnetic Waveguide Analysis

By Evan Lezar and David B. Davidson

Chapter ref: **[lezar]**

► Editor note*: Reduce the number of macros.*

At their core, Maxwell's equations are a set of differential equations describing the interactions between electric and magnetic fields, charges, and currents. These equations provide the tools with which to predict the behaviour of electromagnetic phenomena, giving us the ability to use them in a wide variety of applications, including communication and power generation. Due to the complex nature of typical problems in these fields, numeric methods such as the finite element method are often employed to solve them.

One of the earliest applications of the finite element method in electromagnetics was in waveguide analysis (Davidson, 2005). Since waveguides are some of the most common structures in microwave engineering, especially in areas where high power and low loss are essential (Pozar, 2005), their analysis is still a topic of much interest. This chapter considers the use of FEniCS in the cutoff and dispersion analysis of these structures as well as the analysis of waveguide discontinuities. These types of analysis form an important part of the design and optimisation of waveguide structures for a particular purpose.

The aim of this chapter is to guide the reader through the process followed in implementing solvers for various electromagnetic problems with both cutoff and dispersion analysis considered in depth. To this end a brief introduction of electromagnetic waveguide theory, the mathematical formulation of these problems, and the specifics of their solution using the finite element method are presented in 33.1. This lays the groundwork for a discussion of the details pertaining to the FEniCS implementation of these solvers, covered in 33.2. The translation of the finite element formulation to FEniCS, as well as some post-processing considerations are covered. In 33.3 the solution results for three typical waveguide configurations are presented and compared to analytical or previously published data. This serves to validate the implementation and illustrates the kinds of problems that can be solved. Results for the analysis of H-plane waveguide discontinuities are then presented in 33.4 with two test cases being considered.

33.1 Formulation

As mentioned, in electromagnetics, the behaviour of the electric and magnetic fields are described by Maxwell's equations (Jin, 2002, Smith, 1997). Using these partial differential equations, various boundary value problems can be obtained depending on the problem being solved. In the case of time-harmonic fields, the equation used is the vector Helmholtz wave equation. If the problem is further restricted to a domain surrounded by perfect electrical or magnetic conductors (as is the case in general waveguide problems) the wave equation in terms of the electric field, E, can be written as (Jin, 2002)

$$\nabla \times \frac{1}{\mu_r} \nabla \times \mathbf{E} - k_o^2 \epsilon_r \mathbf{E} = 0, \text{ in } \Omega, \tag{33.1}$$

subject to the boundary conditions

$$\hat{\mathbf{n}} \times \mathbf{E} = 0 \text{ on } \Gamma_e \tag{33.2}$$

$$\hat{\mathbf{n}} \times \nabla \times \mathbf{E} = 0 \text{ on } \Gamma_m, \tag{33.3}$$

with Ω representing the interior of the waveguide and Γ_e and Γ_m electric and magnetic walls respectively. μ_r and ϵ_r are the relative magnetic permeability and electric permittivity respectively. These are material parameters that may be position dependent but only the isotropic case is considered here. k_o is the operating wavenumber which is related to the operating frequency (f_o) by the expression

$$k_o = \frac{2\pi f_o}{c_0}, \tag{33.4}$$

with c_0 the speed of light in free space. This boundary value problem (BVP) can also be written in terms of the magnetic field (Jin, 2002), but as the discussions following are applicable to both formulations this will not be considered here.

If the guide is sufficiently long, and the z-axis is chosen parallel to its central axis as shown in Figure 33.1, then z-dependence of the electric field can be assumed to be of the form $e^{-\gamma z}$ with $\gamma = \alpha + j\beta$ a complex propagation constant (Pelosi et al., 1998, Pozar, 2005). Making this assumption and splitting the electric field into transverse ($\mathbf{E}_t$) and axial ($\hat{\mathbf{z}}E_z$) components, results in the following expression for the field

$$\mathbf{E}(x, y, z) = [\mathbf{E}_t(x, y) + \hat{\mathbf{z}}E_z(x, y)]e^{-\gamma z}, \tag{33.5}$$

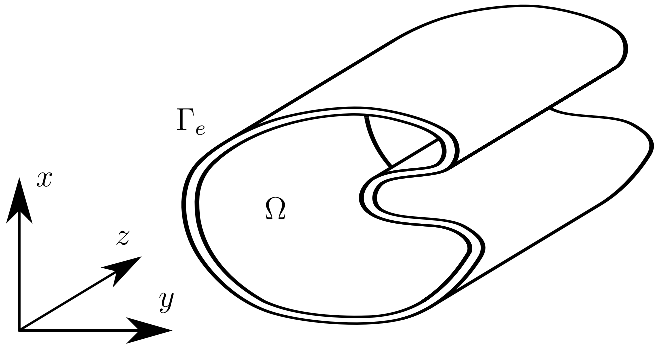


Figure 33.1: A long waveguide with an arbitrary cross-section aligned with the z-axis.

with x and y the Cartesian coordinates in the cross-sectional plane of the waveguide and z the coordinate along the length of the waveguide.

From (33.5) as well as the BVP described by (33.1), (33.2), and (33.3) it is possible to obtain the following variational functional found in many computational electromagnetic texts (Jin, 2002, Pelosi et al., 1998)

$$F(\mathbf{E}) = \frac{1}{2}\int_\Omega \frac{1}{\mu_r}(\nabla_t \times \mathbf{E}_t)\cdot(\nabla_t \times \mathbf{E}_t) - k_o^2\epsilon_r \mathbf{E}_t \cdot \mathbf{E}_t \\ + \frac{1}{\mu_r}(\nabla_t E_z + \gamma \mathbf{E}_t)\cdot(\nabla_t E_z + \gamma \mathbf{E}_t) - k_o^2 \epsilon_r E_z E_z d\Omega, \quad (33.6)$$

with

$$\nabla_t = \frac{\partial}{\partial x}\hat{\mathbf{x}} + \frac{\partial}{\partial y}\hat{\mathbf{y}} \quad (33.7)$$

the transverse del operator.

A number of other approaches have also been taken to this problem. Some, for instance, involve only nodal based elements; some use the longitudinal fields as the working variable, and the problem has also been formulated in terms of potentials, rather than fields. A good summary of these may be found in (**?**)Chapter 9]ZhuCan2006. The approach used here, involving transverse and longitudinal fields, is probably the most widely used in practice.

33.1.1 Waveguide Cutoff Analysis

One of the simplest cases to consider, and often a starting point when testing a new finite element implementation, is waveguide cutoff analysis. When a waveguide is operating at cutoff, the electric field is uniform along the z-axis which corresponds with $\gamma = 0$ in (33.5) (Pozar, 2005). Substituting $\gamma = 0$ into (33.6) yields the following functional

$$F(\mathbf{E}) = \frac{1}{2}\int_\Omega \frac{1}{\mu_r}(\nabla_t \times \mathbf{E}_t)\cdot(\nabla_t \times \mathbf{E}_t) - k_c^2\epsilon_r \mathbf{E}_t \cdot \mathbf{E}_t$$
$$+ \frac{1}{\mu_r}(\nabla_t E_z)\cdot(\nabla_t E_z) - k_c^2 \epsilon_r E_z E_z d\Omega. \quad (33.8)$$

The symbol for the operating wavenumber k_o has been replaced with k_c, indicating that the quantity of interest is now the cutoff wavenumber. This quantity in addition to the field distribution at cutoff are of interest in these kinds of problems. Using two dimensional vector basis functions for the discretisation of the transverse field, and scalar basis functions for the axial components, the minimisation of (33.8) is equivalent to solving the following matrix equation

$$\begin{bmatrix} S_{tt} & 0 \\ 0 & S_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = k_c^2 \begin{bmatrix} T_{tt} & 0 \\ 0 & T_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \quad (33.9)$$

which is in the form of a general eigenvalue problem. Here S_{ss} and T_{ss} represents the stiffness and mass common to finite element literature (Davidson, 2005, Jin, 2002) with the subscripts $_{tt}$ and $_{zz}$ indicating transverse or axial components respectively. The entries of the matrices of (33.9) are defined as

$$(s_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times \mathbf{N}_i)\cdot(\nabla_t \times \mathbf{N}_j) d\Omega, \quad (33.10)$$

$$(t_{tt})_{ij} = \int_\Omega \epsilon_r \mathbf{N}_i \cdot \mathbf{N}_j d\Omega, \quad (33.11)$$

$$(s_{zz})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t M_i)\cdot(\nabla_t M_j) d\Omega, \quad (33.12)$$

$$(t_{zz})_{ij} = \int_\Omega \epsilon_r M_i M_j d\Omega, \quad (33.13)$$

with $\int_\Omega d\Omega$ representing integration over the cross-section of the waveguide and $\mathbf{N}_i$ and M_i representing the i^{th} vector and scalar basis functions respectively.

Due to the block nature of the matrices the eigensystem can be written as two smaller systems

$$[S_{tt}]\{e_t\} = k_{c,TE}^2 [T_{tt}]\{e_t\}, \quad (33.14)$$

$$[S_{zz}]\{e_z\} = k_{c,TM}^2 [T_{zz}]\{e_z\}, \quad (33.15)$$

with $k_{c,TE}$ and $k_{c,TM}$ corresponding to the cutoff wavenumbers of the transverse electric (TE) and transverse magnetic (TM) modes respectively. The eigenvectors ($\{e_t\}$ and $\{e_z\}$) of the systems are the coefficients of the vector and scalar basis functions, allowing for the calculation of the transverse and axial field distributions associated with a waveguide cutoff mode.

33.1.2 Waveguide Dispersion Analysis

In the case of cutoff analysis discussed in 33.1.1, one attempts to obtain the value of $k_o^2 = k_c^2$ for a given propagation constant γ, namely $\gamma = 0$. For most waveguide design applications however, k_o is specified and the propagation constant is calculated from the resultant eigensystem (Jin, 2002, Pelosi et al., 1998). This calculation can be simplified somewhat by making the following substitution into (33.6)

$$\mathbf{E}_{t,\gamma} = \gamma \mathbf{E}_t, \tag{33.16}$$

which results in the modified functional

$$\begin{aligned} F(\mathbf{E}) = \frac{1}{2}\int_\Omega \frac{1}{\mu_r}(\nabla_t \times \mathbf{E}_{t,\gamma}) \cdot (\nabla_t \times \mathbf{E}_{t,\gamma}) - k_o^2 \epsilon_r \mathbf{E}_{t,\gamma} \cdot \mathbf{E}_{t,\gamma} \\ - \gamma^2 \left[\frac{1}{\mu_r}(\nabla_t E_z + \mathbf{E}_{t,\gamma}) \cdot (\nabla_t E_z + \mathbf{E}_{t,\gamma}) - k_o^2 \epsilon_r E_z E_z \right] d\Omega. \end{aligned} \tag{33.17}$$

Using the same discretisation as for cutoff analysis discussed in 33.1.1, the matrix equation associated with the solution of the variational problem is given by

$$\begin{bmatrix} A_{tt} & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = \gamma^2 \begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \tag{33.18}$$

with

$$A_{tt} = S_{tt} - k_o^2 T_{tt}, \tag{33.19}$$
$$B_{zz} = S_{zz} - k_o^2 T_{zz}, \tag{33.20}$$

which is in the form of a generalised eigenvalue problem with the eigenvalues corresponding to the square of the complex propagation constant (γ).

The matrices S_{tt}, T_{tt}, S_{zz}, and T_{zz} are identical to those defined in 33.1.1 with entries given by (33.10), (33.11), (33.12), and (33.13) respectively. The entries of the other sub-matrices, B_{tt}, B_{tz}, and B_{zt}, are defined by

$$(b_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r} \mathbf{N}_i \cdot \mathbf{N}_j d\Omega, \tag{33.21}$$
$$(b_{tz})_{ij} = \int_\Omega \frac{1}{\mu_r} \mathbf{N}_i \cdot \nabla_t M_j d\Omega, \tag{33.22}$$
$$(b_{zt})_{ij} = \int_\Omega \frac{1}{\mu_r} \nabla_t M_i \cdot \mathbf{N}_j d\Omega. \tag{33.23}$$

A common challenge in electromagnetic eigenvalue problems such as these is the occurrence of spurious modes (Davidson, 2005). These are non-physical modes that fall in the null space of the $\nabla \times \nabla \times$ operator of (33.1) (Bossavit, 1998) (The issue of spurious modes is not as closed as most computational electromagnetics texts indicate. For a summary of recent work in the applied mathematics literature, written for an engineering readership, see (Fernandes and Raffetto, 2002)).

One of the strengths of the vector basis functions used in the discretisation of the transverse component of the field is that it allows for the identification of these spurious modes (Davidson, 2005, Jin, 2002). In (Lee et al., 1991) a scaling method is proposed to shift the eigenvalue spectrum such that the dominant waveguide mode (usually the lowest non-zero eigenvalue) corresponds with the largest eigenvalue of the new system. Other approaches have also been followed to address the spurious modes. In (Vardapetyan and Demkowicz, 2002), Lagrange mutipliers are used to move these modes from zero to infinity.

In the case of the eigensystem associated with dispersion analysis, the matrix equation of (33.18) is scaled as follows

$$\begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = \frac{\theta^2}{\theta^2 + \gamma^2} \begin{bmatrix} B_{tt} + \frac{A_{tt}}{\theta^2} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \tag{33.24}$$

with $\theta^2 = k_o^2 \mu_r^{(max)} \epsilon_r^{(max)}$ an upper bound on the square of the propagation constant (γ^2) and $\mu_r^{(max)}$ and $\epsilon_r^{(max)}$ the maximum relative permeability and permittivity in the computational domain.

If λ is an eigenvalue of the scaled system of (33.24), then the propagation constant can be calculated as

$$\gamma^2 = \frac{1 - \lambda}{\lambda} \theta^2, \tag{33.25}$$

and thus $\gamma^2 \to \infty$ as $\lambda \to 0$, which moves the spurious modes out of the region of interest.

33.2 Implementation

This section considers the details of the implementation of a FEniCS-based solver for waveguide cutoff mode and dispersion curve problems as described in 33.1.1 and 33.1.2. A number of code snippets illustrate some of the finer points of the implementation.

33.2.1 Formulation

Code Listing 33.1 shows the definitions of the function spaces used in the solution of the problems considered here. Nédélec basis functions of the first kind (`N_v` and `N_u`) are used to approximate the transverse component of the electric field. This ensures that the tangential continuity required at element and material boundaries can be enforced (Jin, 2002). The axial component of the field is modelled using a set of Lagrange basis functions (`M_v`, and `M_u`). Additionally, a discontinuous Galerkin function space is included to allow for the modelling of material parameters such as dielectrics.

Code Listing 33.1: Function spaces and basis functions.

```
V_DG = FunctionSpace(mesh, "DG", 0)
V_N = FunctionSpace(mesh, "Nedelec 1st kind H(curl)", transverse_order)
V_M = FunctionSpace(mesh, "Lagrange", axial_order)

combined_space = V_N + V_L

(N_v, M_v) = TestFunctions(combined_space)
(N_u, M_u) = TrialFunctions(combined_space)
```

In order to deal with material properties, the `Expression` class is subclassed and the `eval()` method overridden. This is illustrated in Code Listing 33.2 where a dielectric with a relative permittivity of $\epsilon_r = 4$ that extends to $y = 0.25$ is shown. This class is then instantiated using the discontinuous Galerkin function space already discussed. For constant material properties (such as the inverse of the magnetic permittivity μ_r, in this case) a JIT-compiled expression is used.

Code Listing 33.2: Material properties and expressions.

```
class HalfLoadedDielectric(Expression):
    def eval(self, values, x):
        if x[1] < 0.25:
            values[0] = 4.0
        else:
            values[0] = 1.0;

e_r = HalfLoadedDielectric(V_DG)
one_over_u_r = Expression("1.0")

k_o_squared = Expression("value", {"value" : 0.0})
theta_squared = Expression("value", {"value" : 0.0})
```

The basis functions declared in Code Listing 33.1 and the desired material property functions are now used to create the forms required for the matrix entries specified in 33.1.1 and 33.1.2. The forms are shown in Code Listing 33.3 and the matrices of (33.9), (33.18), and (33.24) can be assembled using the required combinations of these forms with the right hand side of (33.24), `rhs`, provided as an example. It should be noted that the use of JIT-compiled expressions for operating wavenumber and scaling parameters means that the forms need not be recompiled each time the operating frequency is changed. This is especially beneficial when the calculation of dispersion curves is considered since the same calculation is performed for a range of operating frequencies.

From (33.2) it follows that the tangential component of the electric field must be zero on perfectly electrical conducting (PEC) surfaces. What this means in practice is that the degrees of freedom associated with both the Lagrange and Nédélec basis functions on the boundary must be set to zero since there can be no electric field inside a perfect electrical conductor (Smith, 1997). An example for a PEC surface surrounding the entire computational domain is shown in Code Listing 33.4 as the `ElectricWalls` class. This boundary condition can then be applied to the constructed matrices before solving the eigenvalue systems.

The boundary condition given in (33.3) results in a natural boundary condition for the problems considered and thus it is not necessary to explicitly enforce it (Pelosi et al., 1998). Such magnetic walls and the symmetry of a problem are often used to decrease the size of the computational domain although this does limit the solution obtained to even modes (Jin, 2002).

Once the required matrices have been assembled and the boundary conditions applied, the resultant eigenproblem can be solved. This can be done by outputting the matrices and solving the problem externally, or by making use of the eigensolvers provided by SLEPc that can be integrated into the FEniCS package.

33.2.2 Post-Processing

After the eigenvalue system has been solved and the required eigenpair chosen, this can be post-processed to obtain various quantities of interest. For the cutoff wavenumber, this is a relatively straight-forward process and only involves simple operations on the eigenvalues of the system. For the calculation of dispersion curves and visualisation of the resultant field components the process is slightly more complex.

Dispersion Curves

For dispersion curves the computed value of the propagation constant (γ) is plotted as a function of the operating frequency (f_o). Since γ is a complex variable, a mapping is required to represent the data on a single two-dimensional graph. This is achieved by choosing the f_o-axis to represent the value $\gamma = 0$, effectively

Code Listing 33.3: Forms for matrix entries.

```
s_tt = one_over_u_r*dot(curl_t(N_v), curl_t(N_u))
t_tt = e_r*dot(N_v, N_u)

s_zz = one_over_u_r*dot(grad(M_v), grad(M_u))
t_zz = e_r*M_v*M_u

b_tt = one_over_u_r*dot(N_v, N_u)
b_tz = one_over_u_r*dot(N_v, grad(M_u))
b_zt = one_over_u_r*dot(grad(M_v), N_u)

a_tt = s_tt - k_o_squared*t_tt
b_zz = s_zz - k_o_squared*t_zz

rhs = b_tt + b_tz + b_zt + b_zz + 1.0/theta_squared*a_tt
```

Code Listing 33.4: Boundary conditions.

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

zero = Expression(("0.0","0.0","0.0"))
dirichlet_bc = DirichletBC(combined_space, zero, ElectricWalls())
```

dividing the $\gamma - f_o$ plane into two regions. The region above the f_o-axis is used to represent the magnitude of the imaginary part of γ, whereas the real part falls in the lower region. A mode that propagates along the guide for a given frequency will thus lie in the upper half-plane of the plot and a complex mode will be represented by a data point above and below the f_o-axis. This procedure is followed in (Pelosi et al., 1998) and other literature and allows for quick comparisons and validation of results.

Field Visualisation

In order to visualise the fields associated with a given solution, the basis functions need to be weighted with coefficients corresponding to the entries in an eigenvector obtained from one of the eigenvalue problems. In addition, the transverse or axial components of the field may need to be extracted. An example for plotting the transverse and axial components of the field is given in Code Listing 33.5. Here the variable `x` assigned to the function vector is one of the eigenvectors obtained by solving the eigenvalue problem.

Code Listing 33.5: Extraction and visualisation of transverse and axial field components.

```
f = Function(combined_space, x)

(transverse, axial) = f.split()

plot(transverse)
plot(axial)
```

The `eval()` method of the `transverse` and `axial` functions can also be called in order to evaluate the functions at a given spatial coordinate, allowing for further visualisation or post-processing options.

33.3 Examples

The first of the examples considered is the canonical one of a hollow waveguide, which has been covered in a multitude of texts on the subject (Davidson, 2005, Jin, 2002, Pelosi et al., 1998, Pozar, 2005). Since the analytical solutions for this structure are known, it provides an excellent benchmark and is a typical starting point for the validation of a computational electromagnetic solver for solving waveguide problems.

The second and third examples are a partially filled rectangular guide and a shielded microstrip line on a dielectric substrate, respectively. In each case results are compared to published results from the literature as a means of validation.

33.3.1 Hollow Rectangular Waveguide

Figure 33.2 shows the cross section of a hollow rectangular waveguide. For the purpose of this chapter a guide with dimensions $a = 1$m and $b = 0.5$m is considered. The analytical expressions for the electric field components of a hollow

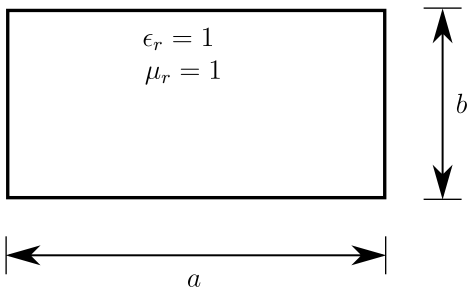


Figure 33.2: A diagram showing the cross section and dimensions of a hollow rectangular waveguide.

rectangular guide with width a and height b are given by (Pozar, 2005)

$$E_x = \frac{n}{b} A_{mn} \cos\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right), \tag{33.26}$$

$$E_y = \frac{-m}{a} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \cos\left(\frac{n\pi y}{b}\right), \tag{33.27}$$

for the TE$_{mn}$ mode, whereas the z-directed electric field for the TM$_{mn}$ mode has the form (Pozar, 2005)

$$E_z = B_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right), \tag{33.28}$$

with A_{mn} and B_{mn} constants for a given mode. In addition, the propagation constant, γ, has the form

$$\gamma = \sqrt{k_o^2 - k_c^2}, \tag{33.29}$$

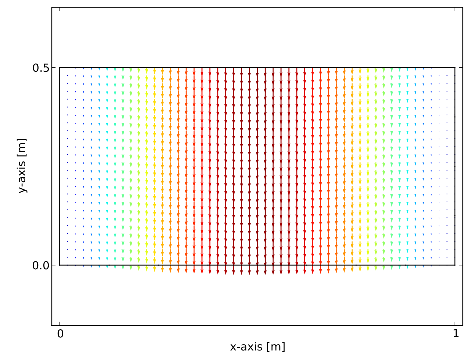
with k_o the operating wavenumber dependent on the operating frequency, and

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2, \tag{33.30}$$

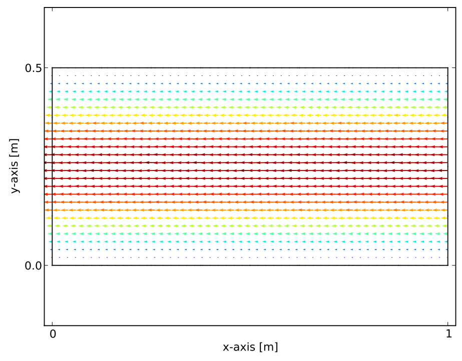
the analytical solution for the square of the cutoff wavenumber for both the TE$_{mn}$ and TM$_{mn}$ modes.

Cutoff Analysis

Figure 33.3 shows the first two calculated TE cutoff modes for the hollow rectangular guide, with the first two TM cutoff modes being shown in Figure 33.4. The solution is obtained with 64 triangular elements and second order basis functions in the transverse as well as the axial discretisations.

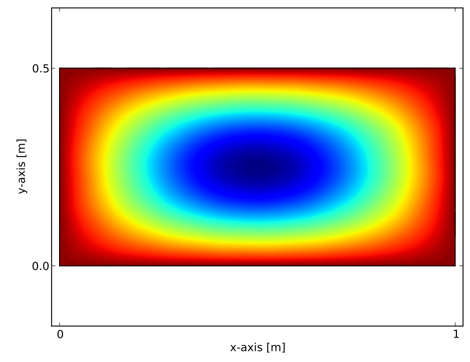


(a) TE_{10} mode.

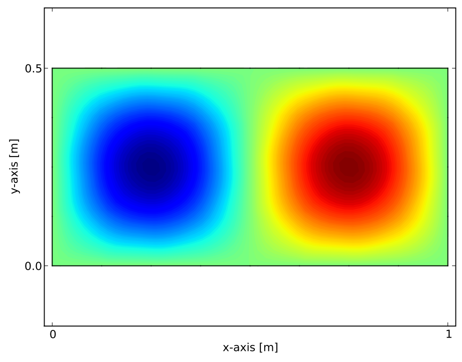


(b) TE_{01} mode.

Figure 33.3: The first two calculated TE cutoff modes of a 1 m $\times$ 0.5 m hollow rectangular waveguide.



(a) TM_{11} mode.



(b) TM_{21} mode.

Figure 33.4: The first two calculated TM cutoff modes of a 1 m $\times$ 0.5 m hollow rectangular waveguide.

Table 33.1 gives a comparison of the calculated and analytical values for the square of the cutoff wavenumber of a number of modes for a hollow rectangular guide. As can be seen from the table, there is excellent agreement between the values.

Table 33.1: Comparison of analytical and calculated cutoff wavenumber squared (k_c^2) for various TE and TM modes of a 1 m $\times$ 0.5 m hollow rectangular waveguide.

Mode	Analytical [m^{-2}]	Calculated [m^{-2}]	Relative Error
TE$_{10}$	9.8696	9.8696	1.4452e-06
TE$_{01}$	39.4784	39.4784	2.1855e-05
TE$_{20}$	39.4784	39.4784	2.1894e-05
TM$_{11}$	49.3480	49.4048	1.1514e-03
TM$_{21}$	78.9568	79.2197	3.3295e-03
TM$_{31}$	128.3049	129.3059	7.8018e-03

Dispersion Analysis

When considering the calculation of the dispersion curves for the hollow rectangular waveguide, the mixed formulation as discussed in 33.1.2 is used. The calculated dispersion curves for the first 10 modes of the hollow rectangular guide are shown in Figure 33.5 along with the analytical results. For the rectangular guide a number of modes are degenerate with the same dispersion and cutoff properties as predicted by (33.29) and (33.30). This explains the visibility of only six curves. There is excellent agreement between the analytical and computed results.

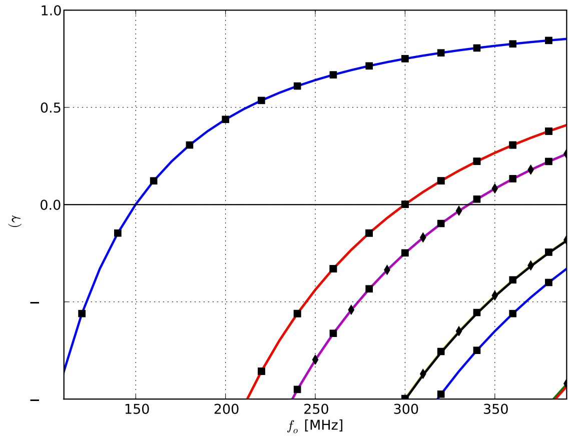


Figure 33.5: Dispersion curves for the first 10 modes of a 1 m $\times$ 0.5 m hollow rectangular waveguide. Markers are used to indicate the analytical results with ■ and ⧫ indicating TE and TM modes respectively.

33.3.2 Half-Loaded Rectangular Waveguide

In some cases, a hollow rectangular guide may not be the ideal structure to use due to, for example, limitations on its dimensions. If the guide is filled with a dielectric material with a relative permittivty $\epsilon_r > 1$, the cutoff frequency of the dominant mode will be lowered. Consequently a loaded waveguide will be mode compact than a hollow guide for the same dominant mode frequency. Furthermore, in many practical applications, such as impedance matching or phase shifting sections, a waveguide that is only partially loaded is used (Pozar, 2005).

Figure 33.6 shows the cross section of such a guide. The guide considered here has the same dimensions as the hollow rectangular waveguide used in the previous section, but its lower half is filled with an $\epsilon_r = 4$ dielectric material.

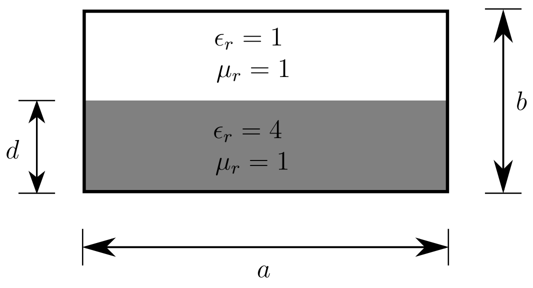
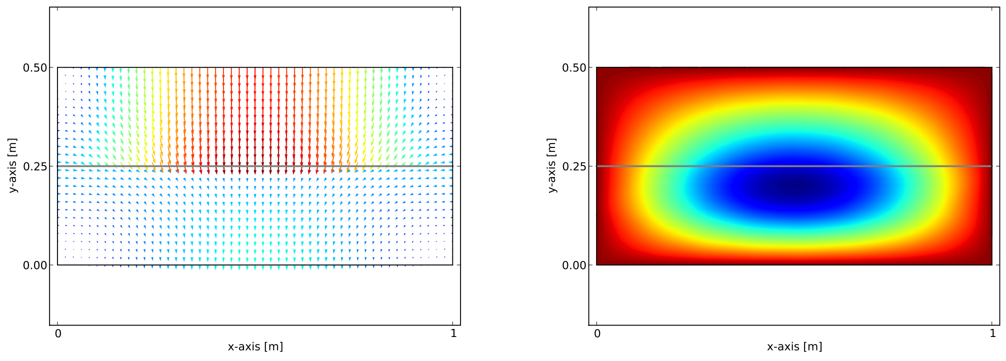


Figure 33.6: A diagram showing the cross section and dimensions of a half-loaded rectangular waveguide. The lower half of the guide is filled with an $\epsilon_r = 4$ dielectric material.

Cutoff Analysis

Figure 33.7 shows the first TE and TM cutoff modes of the half-loaded guide shown in Figure 33.6. Note the concentration of the transverse electric field in the hollow part of the guide. This is due to the fact that the displacement flux, D $=\epsilon$E, must be normally continuous at the dielectric interface (Pozar, 2005, Smith, 1997).



(a) First TE mode. (b) First TM mode.

Figure 33.7: The first calculated TE and TM cutoff modes of a 1 m $\times$ 0.5 m rectangular waveguide with the lower half of the guide filled with an $\epsilon_r = 4$ dielectric.

Dispersion Analysis

The dispersion curves for the first 8 modes of the half-loaded waveguide are shown in Figure 33.8 with results for the first 4 modes from (Jin, 2002) provided as reference. Here it can be seen that the cutoff frequency of the dominant mode has decreased and there is no longer the same degeneracy in the modes

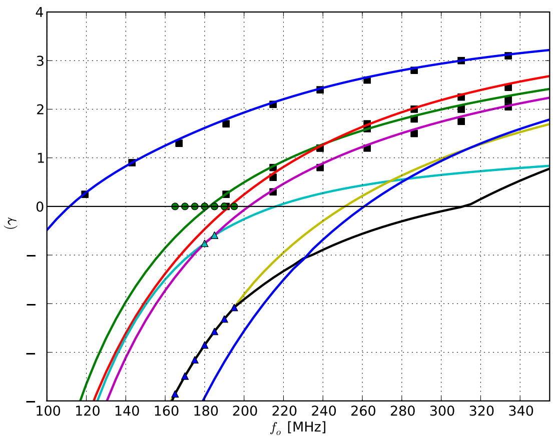


Figure 33.8: Dispersion curves for the first 8 modes of a 1 m $\times$ 0.5 m rectangular waveguide with its lower half filled with an $\epsilon_r = 4$ dielectric material. Reference values for the first 4 modes from (Jin, 2002) are shown as ■. The presence of complex mode pairs are indicated by ▲ and •.

when compared to the hollow guide of the same dimensions. In addition, there are complex modes present as a result of the fourth and fifth as well as the sixth and seventh modes occurring as conjugate pairs at certain points in the spectrum. It should be noted that the imaginary parts of these conjugate pairs are very small and thus the • markers in Figure 33.8 appear to fall on the f_o-axis. These complex modes are discussed further in 33.3.3.

33.3.3 Shielded Microstrip

Microstrip line is a very popular type of planar transmission line, primarily due to the fact that it can be constructed using photolithographic processes and integrates easily with other microwave components (Pozar, 2005). Such a structure typically consists of a thin conducting strip on a dielectric substrate above a ground plane. In addition, the strip may be shielded by enclosing it in a PEC box to reduce electromagnetic interference. A cross section of a shielded microstrip line is shown in Figure 33.9 with the thickness of the strip, t, exaggerated for clarity. The dimensions used to obtain the results discussed here are given in Table 33.2.

Cutoff Analysis

Since the shielded microstrip structure consists of two conductors, it supports a dominant transverse electromagnetic (TEM) wave that has no axial component of the electric or magnetic field (Pozar, 2005). Such a mode has a cutoff wavenum-

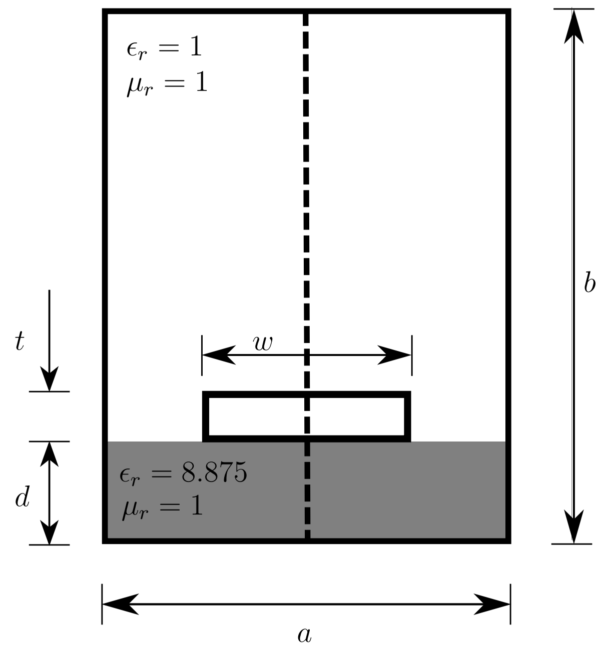


Figure 33.9: A diagram showing the cross section and dimensions of a shielded microstrip line. The microstrip is etched on a dielectric material with a relative permittivity of $\epsilon_r = 8.75$. The plane of symmetry is indicated by a dashed line and is modelled as a magnetic wall in order to reduce the size of the computational domain.

ber of zero and thus propagates for all frequencies (Jin, 2002, Pelosi et al., 1998). The cutoff analysis of this structure is not considered here explicitly. The cutoff wavenumbers for the higher order modes (which are hybrid TE-TM modes (Pozar, 2005)) can however be determined from the dispersion curves by the intersection of a curve with the f_o-axis.

Dispersion Analysis

The dispersion analysis presented in (Pelosi et al., 1998) is repeated here for validation with the resultant curves shown in Figure 33.10. As is the case with the half-loaded guide, the results calculated with FEniCS agree well with pre-

Table 33.2: Dimensions for the shielded microstrip line considered here. Definitions for the symbols are given in Figure 33.9.

	Dimension [mm]
a, b	12.7
d, w	1.27
t	0.127

viously published results. In the figure it is shown that for certain parts of the frequency range of interest, modes six and seven have complex propagation constants. Since the matrices in the eigenvalue problem are real valued, the complex eigenvalues – and thus the propagation constants – must occur in complex conjugate pairs as is the case here and reported earlier in (Huang and Itoh, 1988). These conjugate propagation constants are associated with two equal modes propagating in opposite directions along the waveguide and thus resulting in zero energy transfer. It should be noted that for lossy materials (not considered here), complex modes are expected but do not necessarily occur in conjugate pairs (Pelosi et al., 1998).

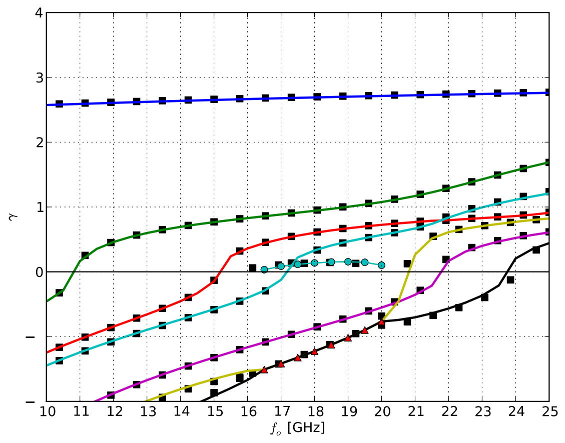


Figure 33.10: Dispersion curves for the first 7 even modes of shielded microstrip line using a magnetic wall to enforce symmetry. Reference values from (Pelosi et al., 1998) are shown as ■. The presence of complex mode pairs are indicated by ▲ and •.

33.4 Analysis of Waveguide Discontinuities

Although this chapter focuses on eigenvalue type problems related to waveguides, the use of FEniCS in waveguide analysis is not limited to such problems. This section briefly introduces the solution of problems relating to waveguide discontinuities as an additional problem class. Applications where the solutions of these problems are of importance to microwave engineers is the design of waveguide filters as well as the analysis and optimisation of bends in a waveguide where properties such as the scattering parameters (S-parameters) of the device are calculated (Pozar, 2005).

The hybrid finite element-modal expansion technique discussed in (Pelosi et al., 1998) is implemented and used to solve problems related to H-plane waveguide

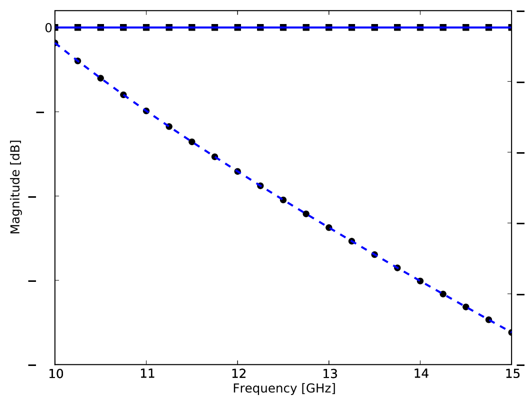


Figure 33.11: Magnitude (solid line) and phase (dashed line) of the transmission coefficient (S_{21}) of a length of rectangular waveguide with dimensions $a = 18.35$mm, $b = 9.175$mm, and $l = 10$mm. The analytical results for the same structure are indicated by markers with ■ and • indicating the magnitude and phase respectively.

discontinuities. For such cases – which are uniform in the vertical (y) direction transverse to the direction of propagation (z) – the problem reduces to a scalar one in two dimensions (Jin, 2002) with the operating variable the y-component of the electric field in the guide. In order to obtain the scattering parameters at each port of the device, the field on the boundary associated with the port is written as a summation of the tangential components of the incoming and outgoing waveguide modes. These modes can either be computed analytically, when a junction is rectangular for example, or calculated with methods such as those discussed in the preceding sections (Martini et al., 2003).

Transmission parameter (S_{21}) results for a length of hollow rectangular waveguide are shown in Figure 33.11. As expected, the length of guide behaves as a fixed value phase shifter (Pelosi et al., 1998) and the results obtained show excellent agreement with the analytical ones.

A schematic for a more interesting example is the H-plane iris shown in Figure 33.12. The figure shows the dimensions of the structure and indicates the port definitions. The boundaries indicated by a solid line is a PEC material. The magnitude and phase results for the S-parameters of the device are given in Figure 33.13 and compared to the results published in (Pelosi et al., 1998) with good agreement between the two sets of data.

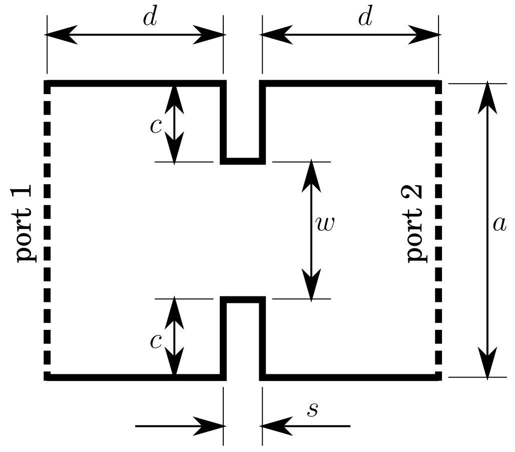
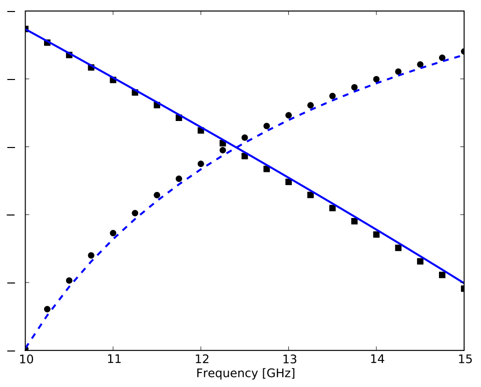


Figure 33.12: Schematic of an H-plane iris in a rectangular waveguide dimensions: $a = 18.35$mm, $c = 4.587$mm, $d = 1$mm, $s = 0.5$mm, and $w = 9.175$mm. The guide has a height of $b = 9.175$mm. The ports are indicated by dashed lines on the boundary of the device.

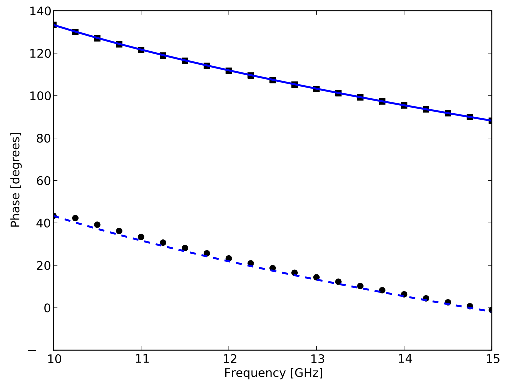
33.5 Conclusion

In this chapter, the solutions of cutoff and dispersion problems associated with electromagnetic waveguiding structures have been implemented and the results analysed. In all cases, the results obtained agree well with previously published or analytical results. This is also the case where the analysis of waveguide discontinuities are considered, and although the solutions shown here are restricted to H-plane waveguide discontinuities, the methods applied are applicable to other classes of problems such as E-plane junctions and full 3D analysis.

This chapter has also illustrated the ease with which complex formulations can be implemented and how quickly solutions can be obtained. This is largely due to the almost one-to-one correspondence between the expressions at a formulation level and the high-level code that is used to implement a particular solution. Even in cases where the required functionality is limited or missing, the use of FEniCS in conjunction with external packages greatly reduces development time.



(a) Magnitude.



(b) Phase.

Figure 33.13: Results for the magnitude and phase of the reflection coefficient (S_{11} – solid line) and transmission coefficient (S_{21} – dashed line) of an H-plane iris in a rectangular waveguide shown in Figure 33.12. Reference results from (Pelosi et al., 1998) are indicated by markers with ■ and • indicating the reflection and transmission coefficient respectively.

CHAPTER 34

Applications in Solid Mechanics

By Kristian B. Ølgaard and Garth N. Wells

Chapter ref: **[oelgaard-1]**

Summarise work on automated modelling for solid mechanics, with application to hyperelasticity, plasticity and strain gradient dependent models. Special attention will be paid the linearisation of function which do come from a finite element space.

CHAPTER 35

Modelling Evolving Discontinuities

By Mehdi Nikbakht and Garth N. Wells

Chapter ref: **[nikbakht]**

Summarise work on automated modelling of PDEs with evolving discontinuities, e.g. cracks.

CHAPTER 36

Block Preconditioning of Systems of PDEs

By Kent-Andre Mardal

Chapter ref: **[block-prec]**

36.1 Introduction

In this chapter we will describe the implementation of block preconditioned Krylov solvers for systems of partial differential equations (PDEs) using the Python interfaces of Dolfin and Trilinos. We remark that an alternative to PyTrilinos is PyAMG (**?**) which can be used together with PyDolfin.

An outline of this paper is as follows: First, we review the abstract theory of constructing preconditioners by considering the differential operators as mappings in properly chosen Sobolev spaces. Second, we will present several examples, namely the Poisson problem, the Stokes problem, the time-dependent Stokes problem and finally a mixed formulation of the Hodge Laplacian. The code examples related to this chapter can be found in FENICSBOOK/src/block-prec. The code examples presented in this chapter differ slightly from the source code, in the sense that import statements, safety checks, command-line arguments, definitions of `Functions` and `Subdomains` are often removed to shorten the presentation.

36.2 Abstract Framework for Constructing Preconditioners

This presentation of preconditioning is largely taken from the review paper (**?**), where a more comprehensive mathematical presentation is given. Consider the following abstract formulation of a linear PDE problem:
Find u in the Hilbert space H such that:

$$\mathcal{A}u = f,$$

where $f \in H^*$ and H^* is the dual space of H. We will assume that the PDE problem is well-posed, i.e., that $\mathcal{A} : H \to H^*$ is a bounded invertible operator in the sense that,

$$\|\mathcal{A}\|_{\mathcal{L}(H,H^*)} \leq C \quad \text{and} \quad \|\mathcal{A}^{-1}\|_{\mathcal{L}(H^*,H)} \leq C.$$

The reader should notice that this operator is bounded only when viewed as an operator from H to H^*. On the other hand, the spectrum of the operator is unbounded. Analogously, discretizations of the operator will typically have condition numbers that increase in powers of h, where h is the characteristic cell size, as the mesh is refined. The remedy for the unbounded spectrum is to introduce a preconditioner. Let the preconditioner B be an operator mapping H^* to H such that

$$\|\mathcal{B}\|_{\mathcal{L}(H^*,H)} \leq C \quad \text{and} \quad \|\mathcal{B}^{-1}\|_{\mathcal{L}(H,H^*)} \leq C.$$

Then $\mathcal{B}\mathcal{A} : H \to H$ and

$$\|\mathcal{B}\mathcal{A}\|_{\mathcal{L}(H,H)} \leq C^2 \quad \text{and} \quad \|(\mathcal{B}\mathcal{A})^{-1}\|_{\mathcal{L}(H,H)} \leq C^2.$$

Hence, the spectrum and therefore the condition number of the preconditioned operator,

$$\kappa(\mathcal{B}\mathcal{A}) = \|\mathcal{B}\mathcal{A}\|_{\mathcal{L}(H,H)} \|(\mathcal{B}\mathcal{A})^{-1}\|_{\mathcal{L}(H,H)} \leq C^4$$

is bounded. Given that the discretized operators $\mathcal{A}_h$ and $\mathcal{B}_h$ are stable, in the sense that the operator norms are bounded, then the condition number of the discrete preconditioned operator, $\kappa(\mathcal{B}_h\mathcal{A}_h)$, will be bounded independent of h. The number of iterations required by a Krylov solver to reach a certain convergence criterion can typically be bounded by the condition number. Hence, when the condition number of the discrete problem is bounded independent of h, the Krylov solvers will have a converges rate independent of h. If then $\mathcal{B}_h$ is similar to $\mathcal{A}_h$ in terms of storage and evaluation, we will then have an *order-optimal* solution algorithm. We remark that it is crucial that $\mathcal{A}_h$ is a stable operator and will illustrate this for the Stokes problem. Finally we will see that $\mathcal{B}_h$ often can be constructed using multigrid techniques.

36.3 Numerical Examples

36.3.1 The Poisson problem with homogeneous Neumann conditions

The Poisson equation with Neumann conditions reads:
Find u such that

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, \\ \frac{\partial u}{\partial n} &= g \text{ on } \partial\Omega. \end{aligned}$$

The variational problem is:
Find $u \in H^1_0 \cap L^2_0$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\partial\Omega} g v \, ds, \quad \forall v \in H^1_0 \cap L^2_0.$$

Let the linear operator $\mathcal{A}$ be defined in terms of the bilinear form,

$$(\mathcal{A}u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx.$$

It is well-known that $\mathcal{A}$ is a bounded invertible operator from $H^1_0 \cap L^2_0$ into its dual space $H^{-1} \cap L^2_0$. Furthermore, it is well-known that one can construct multigrid preconditioners for this operator such that the preconditioner is spectrally equivalent with the inverse of A, independent of the characteristic size of the cells in the mesh (**???**).

In this example, we will construct a multigrid preconditioner based on the algebraic multigrid package ML which is contained in PyTrilinos. Furthermore, we will estimate the eigenvalues of the preconditioned system.

First of all, the ML preconditioner is constructed as follows,

```
from PyTrilinos import Epetra, AztecOO, TriUtils, ML
from dolfin import down_cast, Vector

class MLPreconditioner:
    def __init__(self, A):
        # create the ML preconditioner
        MLList = {
              "max levels"        : 30,
              "output"            : 1,
              "smoother: type"    : "ML symmetric Gauss-Seidel",
              "aggregation: type" : "Uncoupled",
              "ML validate parameter list" : False
        }
        ml_prec = ML.MultiLevelPreconditioner(down_cast(A).mat(), 0)
        ml_prec.SetParameterList(MLList)
        ml_prec.ComputePreconditioner()

    def __mul__(self, b):
        # apply the ML preconditioner
        x = Vector(b.size())
        err = self.ml_prec.ApplyInverse(down_cast(b).vec(),
                                        down_cast(x).vec())
        return x
```

The linear algebra backends uBlas, PETSc and Trilinos all have a wide range of Krylov solvers. Here, we implement these solvers in Python because we would like to store the intermediate variables and used them to compute an estimate of the condition number. The following code shows the implementation of the Conjugate Gradient method using the Python linear algebra interface in Dolfin:

```
def precondconjgrad_eigest(B, A, x, b, tolerance=1.0E-05,
                           relativeconv=False, maxiter=500):

    r = b - A*x
    z = B*r
    d = 1.0*z

    rz = inner(r,z)

    if relativeconv: tolerance *= sqrt(rz)
    iter = 0
    alphas = []
    betas = []
    while sqrt(rz) > tolerance and iter <= maxiter:
        z = A*d
        alpha = rz / inner(d,z)
        x += alpha*d
        r -= alpha*z
        z = B*r
        rz_prev = rz
        rz = inner(r,z)
        beta =  rz / rz_prev
        d = z + beta*d
        iter += 1
        alphas.append(alpha)
        betas.append(beta)

    e = eigenvalue_estimates(alphas, betas)
    return x, e, iter
```

The intermediate variables called alphas and betas can then be used to estimate the condition number of the preconditioned matrix as follows, see e.g. cite. Notice that since the preconditioned CG method converges quite fast when using AMG as a preconditioner, there will be only a small number of alphas and betas and we may therefore use the dense linear algebra tools in NumPy.

```
def eigenvalue_estimates(alphas, betas):
    # eigenvalues estimates in terms of alphas and betas
    import numpy
    from numpy import linalg
    n = len(alphas)
    A = numpy.zeros([n,n])
    A[0,0] = 1/alphas[0]
    for k in range(1, n):
        A[k,k] = 1/alphas[k] + betas[k-1]/alphas[k-1]
        A[k,k-1] = numpy.sqrt(betas[k-1])/alphas[k-1]
        A[k-1,k] = A[k,k-1]
    e,v = linalg.eig(A)
    e.sort()

    return x, e
```

The following code shows the implementation of a Poisson problem solver, using the above mentioned ML preconditioner and CG algorithm. We remark here that it is essential for the convergence of the method that both the start vector and the right-hand side are both in L^2_0. For this reason we subtract the mean value from the right hand-side. The start vector is zero and does therefore have mean value zero.

```
import Krylov
import MLPrec

# use the Epetra backend
parameters["linear_algebra_backend"] = "Epetra"

# Create mesh and finite element
mesh = UnitSquare(10, 10)
V = FunctionSpace(mesh, "CG", 1)

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

# Assemble symmetric matrix and vector
A, b = assemble_system(a,L)

# create solution vector (also used as start vector)
x = b.copy()
x.zero()

# subtract mean value from right hand-side
c = b.array()
c -= sum(c)/len(c)
b[:] = c

# create preconditioner
B = MLPrec.MLPreconditioner(A)
x, e, iter = Krylov.precondconjgrad_eigest(B, A, x, b, 10e-6, True, 100)

print "Number of iterations ", iter
print "Eigenvalues ", e
print "kappa(BA) ", e[len(e)-1]/e[0]
```

In Table 36.1 we list the number of iterations for convergence and the estimated condition number of the preconditioned system based on the code shown above, see also the source code `poisson_neumann.py`.

h	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
κ	1.56	1.26	2.09	1.49	1.20
$\#iterations$	8	8	9	9	8

Table 36.1: The estimated condition number κ and the number of iterations for convergence with respect to various mesh refinements.

36.3.2 The Stokes Problem

Our next example is the Stokes problem,

$$\begin{aligned} -\Delta u - \nabla p &= f \quad \text{in } \Omega, && (36.1)\\ \nabla \cdot u &= 0 \quad \text{in } \Omega, && (36.2)\\ u &= 0 \quad \text{on } \partial\Omega. && (36.3) \end{aligned}$$

The variational form is:
Find $u, p \in H_0^1 \times L_0^2$ such that

$$\int_\Omega \nabla u : \nabla v \, dx + \int_\Omega \nabla \cdot u \, q \, dx + \int_\Omega \nabla \cdot v \, p \, dx = \int_\Omega f \, v \, dx, \quad \forall v, q \in H_0^1 \times L_0^2.$$

Let the linear operator $\mathcal{A}$ be defined as

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix}.$$

where

$$\begin{aligned} (Au, v) &= \int_\Omega \nabla u : \nabla v \, dx, && (36.4)\\ (Bu, q) &= \int_\Omega \nabla \cdot u \, q \, dx, && (36.5) \end{aligned}$$

and B^* is the adjoint of B. Then it is well-known that $\mathcal{A}$ is a bounded operator from $H_0^1 \times L_0^2$ to its dual $H^{-1} \times L_0^2$, see e.g. (**??**). Therefore, we construct a preconditioner, $\mathcal{B} : H^{-1} \times L_0^2 \to H_0^1 \times L_0^2$ defined as

$$\mathcal{B} = \begin{pmatrix} K^{-1} & 0 \\ 0 & L^{-1} \end{pmatrix}.$$

where

$$\begin{aligned} (Ku, v) &= \int_\Omega \nabla u : \nabla v \, dx && (36.6)\\ (Lp, q) &= \int_\Omega p \, q \, dx. && (36.7) \end{aligned}$$

method	h	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
$P_2 - P_1$	iterations	34	43	49	52	58
$P_2 - P_1$	κ	13.3	13.5	13.6	13.6	13.6
$P_2 - P_0$	iterations	27	35	42	47	54
$P_2 - P_0$	κ	6.9	7.9	8.8	9.5	10.3
$P_1 - P_1$	iterations	36	118	200+	200+	200+
$P_1 - P_1$	κ	147	308	696	827	671
$P_1 - P_1$-stab	iterations	29	34	33	33	31
$P_1 - P_1$-stab	κ	11.0	12.3	12.5	12.6	12.7

Table 36.2: The number of iterations for convergence with respect to mesh refinements. The methods $P_2 - P_1$, $P_2 - P_0$, and $P_1 - P_1$-stab are stable, while $P_1 - P_1$ is not.

We refer to (**?**) for a mathematical explanation of the derivation of such preconditioners. Notice that this operator $\mathcal{B}$ is positive in contrast to $\mathcal{A}$. Hence, the preconditioned operator $\mathcal{BA}$ will be indefinite. For both K and L, we use the AMG preconditioner provided by ML/Trilinos as described in the previous example (A simple Jacobi preconditioner would be sufficient for L). For symmetric indefinite problems the *Minimum Residual Method* is the fastest method. Preconditioners of this form has been studied by many (**????**). In Table 36.2, we present the number of iterations needed for convergence and estimates on the condition number κ with respect to different discretization methods and different characteristic cell sizes h. The MinRes iteration is stopped when $(\mathcal{B}_h r_k, r_k)/(\mathcal{B}_h r_0, r_0) \leq 10^{-10}$, where r_k is the residual at iteration k. The condition numbers, κ, were estimated using the CG method on the normal equation. This condition number will always be less than the real condition number and is probably too low for the last columns for the $P_1 - P_1$ method without stabilization. for all the. Notice that for the stable methods $P_2 - P_1$ and $P_2 - P_0$, the number of iterations and the condition number seems to be bounded independently of h. For the unstable $P_1 - P_1$ method, the number of iterations and the condition number increases as h decreases. However, for the stabilized $P_1 - P_1$ method, where the pressure is stabilized by

$$\int_\Omega \nabla \cdot u \, q - \alpha h^2 \, \nabla p \cdot \nabla q \, dx,$$

the number of iterations and the condition number appear to be bounded.

We will now describe the code in detail. In this case, the preconditioner consists of two preconditioners. The following shows how to implement this block preconditioner based on the ML preconditioner defined in the previous example.

```
class SaddlePrec:
    def __init__(self, K, L):
        self.K = K
        self.L = L
        self.K_prec = MLPreconditioner(K)
        self.L_prec = MLPreconditioner(L)
        self.n = K.size(0)
        self.m = L.size(0)
        self.x = Vector(self.n+self.m)

    def __mul__(self, b):

        self.x = Vector(self.n+self.m)
        x = self.x
        n = self.n
        m = self.m

        x[0:n]    = self.K_prec*b[0:n]
        x[n:n+m]  = self.L_prec*b[n:n+m]

        return x
```

The Stokes problem is then specified and solved as follows:

```
mesh = UnitSquare(40,40)
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
mixed = V + Q

f = Constant(mesh, (0,0))
g = Constant(mesh, 0)

u, p = TrialFunctions(mixed)
v, q = TestFunctions(mixed)

a = inner(grad(u), grad(v))*dx + div(u)*q*dx + div(v)*p*dx
L = dot(f, v)*dx

bc_func = BoundaryFunction(V)
bc = DirichletBC(V, bc_func, Boundary())

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

And finally, we create a block preconditioner and solve the problem with the MinRes method.

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

k = inner(grad(u), grad(v))*dx
l = p*q*dx
L0 = dot(v,f)*dx
L1 = q*g*dx

K, b0 = assemble_system(k,L0,bc)
L, b1 = assemble_system(l,L1)

x = Vector(b.size())
x.zero()

B = SaddlePrec(K, L)

x, i, rho  = MinRes.precondMinRes(B, A, x, b, 10e-8, False, 200)
```

We refer to `stokes.py` for the complete code.

36.3.3 The time-dependent Stokes Problem

Our next example is the time-dependent Stokes problem,

$$\begin{aligned} u - k\Delta u - \nabla p &= f \quad \text{in } \Omega, & (36.8)\\ \nabla \cdot u &= 0 \quad \text{in } \Omega, & (36.9)\\ u &= 0 \quad \text{on } \partial\Omega. & (36.10) \end{aligned}$$

Here k is the time stepping parameter.

The variational form is:
Find $u, p \in H_0^1 \times L_0^2$ such that

$$\int_\Omega u\cdot v\,dx + k\int_\Omega \nabla u : \nabla v\,dx + \int_\Omega \nabla\cdot u\,q\,dx + \int_\Omega \nabla\cdot v\,p\,dx = \int_\Omega f\,v\,dx, \quad \forall v, q \in H_0^1 \times L_0^2.$$

Let

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix}.$$

where

$$\begin{aligned} (Au, v) &= \int_\Omega u\cdot v\,dx + k\int_\Omega \nabla u : \nabla v\,dx, & (36.11)\\ (Bu, q) &= \int_\Omega \nabla\cdot u\,q\,dx, & (36.12) \end{aligned}$$

This operator changes character as k varies. For $k = 1$ the problem behaves like Stokes problem, with a non-harmful low order term. However as k approaches zero the problems change to the mixed formulation of a Poisson equation, ie.

$$\begin{aligned} u - \nabla p &= f, \quad \Omega\\ \nabla\cdot u &= 0, \quad \Omega \end{aligned}$$

This problem is not an well-defined operator from $H_0^1 \times L_0^2$ into its dual. Instead, it is a mapping from $H(div) \times L_0^2$ to its dual. However, as pointed out in (**?**) this operator can also be seen as an operator $L^2 \times H^1$ to its dual. In fact, in (**??**) it was shown that A is a bounded operator from $L^2 \cap kH_0^1 \times H^1 \cap L_0^2 + k^{-1}L_0^2$ to its dual space with a bounded inverse. Furthermore, the bounds are uniform in k. Therefore we construct a preconditioner B, such that

$$\mathcal{B} : L^2 \cap kH_0^1 \times H^1 \cap L_0^2 + k^{-1}L_0^2 \to L^2 + k^{-1}H^{-1} \times H^{-1} \cap L_0^2 + kL_0^2.$$

Such a $\mathcal{B}$ can be defined as

$$\mathcal{B} = \begin{pmatrix} K^{-1} & 0 \\ 0 & L^{-1} + M^{-1} \end{pmatrix}.$$

$k\backslash h$	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1.0	13.2	13.5	13.6	13.6	13.6
0.1	12.5	13.2	13.2	13.5	13.6
0.01	10.3	11.0	12.8	13.2	13.4
0.001	7.3	9.1	11.0	12.3	13.0

Table 36.3: The convergence with respect to k and mesh refinements.

where

$$(Ku, v) = \int_\Omega \nabla u : \nabla v \, dx \tag{36.13}$$

$$(Lp, q) = \int_\Omega k^{-1} pq \, dx \tag{36.14}$$

$$(Mp, q) = \int_\Omega \nabla p \cdot \nabla q \, dx. \tag{36.15}$$

Again we refer to (**?**) and references therein, for an overview of the construction of such preconditioners and a more comprehensive mathematical derivation. Preconditioners of this form has been studied by many, c.f. e.g. (**?????**).

Creating the preconditioner in this example is completely analogous to the Stokes example except that we need three matrices based on three bilinear forms:

```
k = dot(u, v)*dx +  k*inner(grad(u), grad(v))*dx
l = kinv*p*q*dx
m = dot(grad(p),grad(q))*dx
L0 = dot(v,f)*dx
L1 = q*g*dx

K, b0 = assemble_system(k,L0,bc)
L, b1 = assemble_system(l,L1)
M, b1 = assemble_system(m,L1)
```

Also the code for the block preconditioner is analogous, except that it is based on three matrices:

```
class SaddlePrec2:
    def __init__(self, K, L, M):
        self.K_prec = MLPrec.MLPreconditioner(K)
        self.L_prec = MLPrec.MLPreconditioner(L)
        self.M_prec = MLPrec.MLPreconditioner(M)
        self.n = K.size(0)
        self.m = L.size(0)

    def __mul__(self, b):

        n = self.n
        m = self.m
        x = Vector(n+m)
        y0 = Vector(m)
        y1 = Vector(m)
        x[0:n]     = self.K_prec*b[0:n]
        y0         = self.L_prec*b[n:n+m]
        y1         = self.M_prec*b[n:n+m]
        x[n:n+m]    = y0 + y1

        return x
```

The complete code can be found in `timestokes.py`

36.3.4 Mixed form of the Hodge Laplacian

The final example is a mixed formulation of the Hodge Laplacian,

$$\nabla \times \nabla \times u - \nabla p = f \quad \text{in } \Omega, \tag{36.16}$$

$$\nabla \cdot u - p = 0 \quad \text{in } \Omega, \tag{36.17}$$

$$u \times n = 0 \quad \text{on } \partial\Omega, \tag{36.18}$$

$$p = 0 \quad \text{on } \partial\Omega. \tag{36.19}$$

The variational form is:
Find $u, p \in H_0(\text{curl}) \times H_0^1$ such that

$$\int_\Omega \nabla \times u \cdot \nabla \times v \, dx - \int_\Omega \nabla p v \, dx = \int_\Omega f v \, dx \quad \forall v \in H_0(\text{curl}) \tag{36.20}$$

$$\int_\Omega u \nabla q \, dx - \int_\Omega pq \, dx = 0 \quad \forall q \in H_0^1. \tag{36.21}$$

$$\tag{36.22}$$

Hence, it is natural to consider preconditioner a for $H(\text{curl})$ problems (in addition to H^1 preconditioners). Such preconditioners have been considered by many c.f. e.g. (**????**). One important observation in these papers is that point-wise smoothers are not appropriate for geometric multigrid methods. Furthermore, for algebraic multigrid methods, extra care has to be taken for the aggregation step (**??**).

Let

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & -C \end{pmatrix},$$

where,

$$(Au, v) = \int_\Omega \nabla \times u \cdot \nabla \times v \, dx, \tag{36.23}$$

$$(Bp, v) = -\int_\Omega \nabla p v \, dx \tag{36.24}$$

$$(Cp, q) = -\int_\Omega pq \, dx \tag{36.25}$$

Then $\mathcal{A} : H_0(\mathrm{curl}) \times H_0^1 \to H^{-1}(\mathrm{curl}) \times H^{-1}$, where $H^{-1}(\mathrm{curl})$ is the dual of $H_0(\mathrm{curl})$.

However, if we for the moment forget about the boundary conditions, we can obtain the Laplacian form by eliminating p from (36.16)-(36.17), i.e.,

$$\nabla \times \nabla \times u - \nabla\nabla \cdot u = f.$$

Hence, the problem is elliptic in nature, although this is not apparent in the mixed formulation. In other words, modulo boundary conditions, $\mathcal{A} : H^1 \times L^2 \to H^{-1} \times L^2$.

To avoid constructing a $H(\mathrm{curl})$ preconditioner we will employ the observation that this is a vector Laplacian. Let the discrete operator be

$$\mathcal{A} = \begin{pmatrix} A_h & B_h^* \\ B_h & -C_h \end{pmatrix},$$

where we assume that the discrete system has been obtained by using a stable finite element method, eg. using lowest order Nedelec elements of first kind **(?)** combined with continuous piecewise linears. We eliminate the pressure to obtain the matrix

$$K_h = A_h + B_h^* C_h^{-1} B_h,$$

A problem here is that C_h^{-1} is a dense matrix, therefore we lump the C_h matrix before inverting it, ie.,

$$L_h = A_h + B_h^* (diag(C_h))^{-1} B_h,$$

The matrix L_h is in some sense a vector Laplacian, incorporating the mixed discretization technique, that is appropriate to build an AMG preconditioner upon. To test the efficiency of this preconditioner compared with more straightforward applications of AMG, we compare a couple of different problems. First, we test the preconditioners for the A_h and the L_h operators, i.e., we estimate the condition number for the systems $P_1 A_h$ and $P_2 L_h$, where P_1 and P_2 is simply the algebraic multigrid preconditioners for A_h and L_h, respectively. Then we test the preconditioners

$$\mathcal{B}_1 = \begin{pmatrix} A_h & 0 \\ 0 & D_h \end{pmatrix}.$$

Here, D_h is a discrete Laplacian. The other preconditioner is

$$\mathcal{B}_2 = \begin{pmatrix} L_h & 0 \\ 0 & C_h \end{pmatrix}.$$

In Table 36.4 we list the estimated condition numbers on various mesh refinements on the unitcube.

The main problem in this example is the construction of the preconditioner for L_h. This preconditioner is based on a matrix which is constructed by several

h	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
P_1A_h	7.5	20.2	78.4	318.5	1206.3
P_2L_h	1.5	2.1	5.8	20.4	78.0
$\mathcal{B}_1\mathcal{A}$	3.9	6.7	17.6	58.6	-
$\mathcal{B}_2\mathcal{A}$	4.3	8.7	33.6	30.0	-

Table 36.4: The estimated condition number κ and the number of iterations for convergence with respect to various mesh refinements.

matrix-matrix products and Dolfin does not provide functionality for this. However, Epetra/Trilinos have this functionality. To be able to use the functionality of Epetra/Trilinos we down_cast the Dolfin `EpetraMatrix` to its underlying type `Epetra_FECrsMatrix`. We may then use PyTrilions. The following code demonstrate how to perform matrix matrix products, mass lump inversion etc. using PyTrilinos.

```
V = FunctionSpace(mesh, "N1curl", 1)
Q = FunctionSpace(mesh, "CG", 1)

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

aa = dot(u, v)*dx + dot(curl(v), curl(u))*dx
bb = dot(grad(p), v)*dx
ff = q*p*dx

AA = assemble(aa)
BB = assemble(bb)
BF = assemble(bb)
FF = assemble(ff)

AA_epetra = down_cast(AA).mat()
BB_epetra = down_cast(BB).mat()
BF_epetra = down_cast(BF).mat()
FF_epetra = down_cast(FF).mat()

ff_vec = Epetra.Vector(FF_epetra.RowMap())
FF_epetra.InvRowSums(ff_vec)
BF_epetra.RightScale(ff_vec)

CC = Epetra.FECrsMatrix(Epetra.Copy, AA_epetra.RowMap(), 100)
err = EpetraExt.Multiply(BB_epetra, False, BF_epetra, True, CC)
DD = EpetraMatrix(CC)

EE = assemble(aa, DD, reset_sparsity=False, add_values=True)
```

The complete code can be found in `hodge8.py` and `hodge9.py`.

36.4 Conclusion

In this chapter we have demonstrated that advanced solution algorithms can we developed relatively easily by using the Python interfaces of Dolfin and Trilinos.

The Python linear algebra interface in Dolfin allow us to write Krylov solvers and customize them in the language which these algorithms are typically expressed in books. Furthermore, it is relatively simple to employ state–of–the–art algebraic multigrid algorithms in Python using Trilinos.

In this chapter we have shown the implementation of block preconditioners for a few selected problems. Block preconditioners have been used in a varity of applications, we refer to (**?**) and the references therein for a more complete discussion on this topic.

CHAPTER 37

Automatic Calibration of Depositional Models

By Hans Joachim Schroll

Chapter ref: **[schroll]**

A novel concept for calibrating depositional models is presented. In this approach transport coefficients are determined from well output measurements. Finite element implementation of the multi–lithology models and their duals is automated by the FEniCS project DOLFIN using a python interface.

37.1 Issues in dual lithology sedimentation

Different types of forward computer models are being used by sedimentologists and geomorphologists to simulate the process of sedimentary deposition over geological time periods. The models can be used to predict the presence of reservoir rocks and stratigraphic traps at a variety of scales. State–of–the–art advanced numerical software provides accurate approximations to the mathematical model, which commonly is expressed in terms of a nonlinear diffusion dominated PDE system. The potential of todays simulation software in industrial applications is limited however, due to major uncertainties in crucial material parameters that combine a number of physical phenomena and therefore are difficult to quantify. Examples of such parameters are diffusive transport coefficients.

The idea in this contribution is to calibrate uncertain transport coefficients to direct observable data, like well measurements from a specific basin. In this approach the forward evolution process, mapping data to observations, is reversed to determine the data, i.e., transport coefficients. Mathematical tools and numerical algorithms are applied to automatically calibrate geological models to actual observations — a critical but so far missing link in forward depositional modeling.

Automatic calibration, in combination with stochastic modeling, will boost the applicability and impact of modern numerical simulations in industrial applications.

37.2 A multidimensional sedimentation model

Submarine sedimentation is an evolution process. By flow into the basin, sediments build up and evolve in time. The evolution follows geophysical laws, expressed as diffusive PDE models. The following system is a multidimensional version of the dual lithology model by Rivenæs (Rivenæs, 1992, 1993)

$$\begin{pmatrix} A & s \\ -A & 1-s \end{pmatrix} \begin{pmatrix} s \\ h \end{pmatrix}_t = \nabla \cdot \begin{pmatrix} \alpha s \nabla h \\ \beta(1-s)\nabla h \end{pmatrix} \text{ in } [0,T]\times B \ . \tag{37.1}$$

Here h denotes the thickness of a layer of deposit and s models the volume fraction for the sand lithology. Consequently, $1-s$ is the fraction for mud. The system is driven by fluxes anti proportional to the flow rates $s\nabla h$ and $(1-s)\nabla h$ resulting in a diffusive, but incompletely parabolic, PDE system. The domain of the basin is denoted by B. Parameters in the model are: The transport layer thickness A and the diffusive transport coefficients α, β.

For a forward in time simulation, the system requires initial and boundary data. At initial time, the volume fraction s and the layer thickness h need to be specified. According to geologists, such data can be reconstructed by some kind of "back stripping". Along the boundary of the basin, the flow rates $s\nabla h$ and $(1-s)\nabla h$ are given.

37.3 An inverse approach

The parameter–to–observation mapping $R : (\alpha, \beta) \mapsto (s, h)$ is commonly referred to as the forward problem. In a basin direct observations are only available at wells. Moreover, from the age of the sediments, their history can be reconstructed. Finally, well–data is available in certain well areas $W \subset B$ and backward in time.

The objective of the present investigation is to determine transport coefficients from observed well–data and in that way, to calibrate the model to the data. This essentially means to invert the parameter–to–observation mapping. Denoting observed well–data by $(\widetilde{s}, \widetilde{h})$, the goal is to minimize the output functional

$$J(\alpha,\beta) = \frac{1}{|W|}\int_0^T \int_W (\widetilde{s}-s)^2 + (\widetilde{h}-h)^2 \, dx \, dt \tag{37.2}$$

with respect to the transport coefficients α and β.

In contrast to the "direct inversion" as described by Imhof and Sharma (Imhof and Sharma, 2007), which is considered impractical, we do not propose to invert the time evolution of the diffusive depositional process. We actually use the forward–in–time evolution of sediment layers in a number of wells to calibrate transport coefficients. Via the calibrated model we can simulate the basin and reconstruct its historic evolution. By computational mathematical modeling, the local data observed in wells determines the evolution throughout the entire basin.

37.4 The Landweber algorithm

In a slightly more abstract setting, the task is to minimize an objective functional J which implicitly depends on the parameters p via u subject to the constraint that u satisfies some PDE model; a PDE constrained minimization problem: Find p such that $J(p) = J(u(p)) = \min$ and $\mathrm{PDE}(u, p) = 0$.

Landweber's steepest decent algorithm **(?)** iterates the following sequence until convergence:

1. Solve $\mathrm{PDE}(u^k, p^k) = 0$ for u^k.
2. Evaluate $d^k = -\nabla_p J(p^k)/\|\nabla_p J(p^k)\|$.
3. Update $p^{k+1} = p^k + \Delta p^k d^k$.

Note that the search direction d^k, the negative gradient, is the direction of steepest decent. To avoid scale dependence, the search direction is normed.

The increment Δp^k is determined by a one dimensional line search algorithm, minimizing a locally quadratic approximation to J along the line $p^k + \gamma d^k$. We use the ansatz

$$J(p^k + \gamma d^k) = a\gamma^2 + b\gamma + c \ , \quad \gamma \in \mathbb{R} \ .$$

The extreme value of this parabola is located at

$$\gamma_e = -\frac{b}{2a} \ . \tag{37.3}$$

To determine γ_e, the parabola is fitted to the local data. For example b is given by the gradient

$$J_\gamma(p^k) = b = \nabla_p J(p^k) \cdot d^k = -\|\nabla_p J(p^k)\| \ .$$

To find a, another gradient of J along the line $p^k + \gamma d^k$ is needed. To avoid an extra evaluation, we project p^{k-1} onto the line and approximate the directional derivative

$$J_\gamma(p^k - \gamma^k d^k) \approx \nabla_p J(p^{k-1}) \cdot d^k \ . \tag{37.4}$$

Note that this approximation is exact if two successive directions d^{k-1} and d^k are in line. Elementary geometry yields $\gamma^k = \gamma^{k-1}\cos\varphi$, $\cos\varphi = d^{k-1}\cdot d^k$ and $\gamma^k = \gamma^{k-1}\cdot d^{k-1}\cdot d^k$. From (37.4) we find

$$-2a = \frac{(\nabla_p J(p^{k-1}) - \nabla_p J(p^k))\cdot d^k}{\gamma^{k-1}\cdot d^{k-1}\cdot d^k}$$

Thus, the increment (37.3) evaluates as

$$\Delta p^k = \gamma_e = \frac{\nabla_p J(p^k)\cdot\nabla_p J(p^{k-1})}{\nabla_p J(p^k)\cdot\nabla_p J(p^{k-1}) - \|\nabla_p J(p^k)\|^2}\cdot\frac{\|\nabla_p J(p^k)\|}{\|\nabla_p J(p^{k-1})\|}\cdot\gamma^{k-1} \quad .$$

37.5 Evaluation of gradients by duality arguments

Every single step of Landweber's algorithm requires the simulation of a time dependent, nonlinear PDE system and the evaluation of the gradient of the objective functional. The most common approach to numerical derivatives, via finite differences, is impractical for complex problems: Finite difference approximation would require to perform $n+1$ forward simulations in n parameter dimensions. Using duality arguments however, n nonlinear PDE systems can be replaced by one linear, dual problem. After all, J is evaluated by one forward simulation of the nonlinear PDE model and the complete gradient ∇J is obtained by one (backward) simulation of the linear, dual problem. Apparently, one of the first references to this kind of duality arguments is (Chavent and Lemmonier, 1974).

The concept is conveniently explained for a scalar diffusion equation

$$u_t = \nabla\cdot(\alpha\nabla u) \quad .$$

As transport coefficients may vary throughout the basin, we allow for a piecewise constant coefficient

$$\alpha = \begin{cases} \alpha_1 & x\in B_1 \\ \alpha_2 & x\in B_2 \end{cases} \quad .$$

Assuming no flow along the boundary and selecting a suitable test function ϕ, the equation in variational form reads

$$\mathcal{A}(u,\phi) := \int_0^T\int_B u_t\phi + \alpha\nabla u\cdot\nabla\phi \mathrm{d}x\,\mathrm{d}t = 0 \quad .$$

Taking an derivative $\partial/\partial\alpha_i$, $i = 1,2$ under the integral sign, we find

$$\mathcal{A}(u_{\alpha_i},\phi) = \int_0^T\int_B u_{\alpha_i,t}\phi + \alpha\nabla u_{\alpha_i}\cdot\nabla\phi \mathrm{d}x\,\mathrm{d}t = -\int_0^T\int_{B_i}\nabla u\cdot\nabla\phi \mathrm{d}x\,\mathrm{d}t \quad . \tag{37.5}$$

The corresponding derivative of the output functional $J = \int_0^T \int_W (u-d)^2 \mathrm{d}x\,\mathrm{d}t$ reads

$$J_{\alpha_i} = 2\int_0^T \int_W (u-d)u_{\alpha_i} \mathrm{d}x\,\mathrm{d}t \ , \quad i=1,2 \ .$$

The trick is to define a dual problem

$$\mathcal{A}(\phi,\omega) = 2\int_0^T \int_W (u-d)\phi \mathrm{d}x\,\mathrm{d}t$$

such that $\mathcal{A}(u_{\alpha_i},\omega) = J_{\alpha_i}$ and by using the dual solution ω in (37.5)

$$\mathcal{A}(u_{\alpha_i},\omega) = J_{\alpha_i} = -\int_0^T \int_{B_i} \nabla u \cdot \nabla \omega \mathrm{d}x\,\mathrm{d}t \ , \quad i=1,2 \ .$$

In effect, the desired gradient ∇J is expressed in terms of primal– and dual solutions. In this case the dual problem reads

$$\int_0^T \int_B \phi_t \omega + \alpha \nabla\phi \cdot \nabla\omega \mathrm{d}x\,\mathrm{d}t = 2\int_0^T \int_W (u-d)\phi \mathrm{d}x\,\mathrm{d}t \ , \qquad (37.6)$$

which in strong form appears as a backward in time heat equation with zero terminal condition

$$-\omega_t = \nabla\cdot(\alpha\nabla\omega) + 2(u-d)|_W \ .$$

Note that this dual equation is linear and entirely driven by the data mismatch in the well. With perfectly matching data $d = u|_W$, the dual solution is zero.

Along the same lines of argumentation one derives the multilinear operator to the depositional model (37.1)

$$\begin{aligned}&\mathcal{A}(u,v)(\phi,\psi) = \\ &\int_0^T \int_B (Au_t + uh_t + sv_t)\,\phi + \alpha u \nabla h \cdot \nabla\phi + \alpha s \nabla v \cdot \nabla \phi \mathrm{d}x\,\mathrm{d}t \\ &+ \int_0^T \int_B (-Au_t - uh_t(1-s)v_t)\,\psi - \beta u \nabla h \cdot \nabla \psi + \beta(1-s)\nabla v \cdot \nabla\psi \mathrm{d}x\,\mathrm{d}t \ .\end{aligned}$$

The dual system related to the well output functional (37.2) reads

$$\mathcal{A}(\phi,\psi)(\omega,\nu) = 2\int_0^T \int_W (s-\widetilde{s})\phi + (h-\widetilde{h})\psi \mathrm{d}x\,\mathrm{d}t \ .$$

By construction it follows $\mathcal{A}(s_p,h_p)(\omega,\nu) = J_p(\alpha,\beta)$. Given both primal and dual solutions, the gradient of the well output functional evaluates as

$$\begin{aligned} J_{\alpha_i}(\alpha,\beta) &= -\int_0^T \int_{B_i} s\nabla h \cdot \nabla\omega \mathrm{d}x\,\mathrm{d}t \ , \\ J_{\beta_i}(\alpha,\beta) &= -\int_0^T \int_{B_i} (1-s)\nabla h \cdot \nabla\nu \mathrm{d}x\,\mathrm{d}t \ . \end{aligned}$$

A detailed derivation including non zero flow conditions is given in (Schroll, 2008). For completeness, not for computation(!), we state the dual system in strong form

$$-A(\omega-\nu)_t + h_t(\omega-\nu) + \alpha\nabla h\cdot\nabla\omega = \beta\nabla h\cdot\nabla\nu + \frac{2}{|W|}(s-\widetilde{s})\Big|_W$$
$$-(s\omega+(1-s)\nu)_t = \nabla\cdot(\alpha s\nabla\omega + \beta(1-s)\nabla\nu) + \frac{2}{|W|}(h-\widetilde{h})\Big|_W \ .$$

Obviously the system is linear and driven by the data mismatch at the well. It always comes with zero terminal condition and no flow conditions along the boundary of the basin. Thus, perfectly matching data results in a trivial dual solution.

37.6 Aspects of the implementation

The FEniCS project DOLFIN (Logg and Wells, 2009) automates the solution of PDEs in variational formulation and is therefore especially attractive for implementing dual problems, which are derived in variational form. In this section the coding of the dual diffusion equation (37.6) is illustrated. Choosing a test function supported in $[t_n, t_{n+1}]\times B$ the weak form reads

$$-\int_{t_n}^{t_{n+1}}\int_B \omega_t\phi + \alpha\nabla\omega\cdot\nabla\phi \mathrm{d}x\,\mathrm{d}t = 2\int_{t_n}^{t_{n+1}}\int_W (u-d)\phi\mathrm{d}x\,\mathrm{d}t \ .$$

Trapezoidal rule time integration gives

$$\begin{aligned}&-\int_B(\omega^{n+1}-\omega^n)\phi\mathrm{d}x + \frac{\Delta t}{2}\int_B \alpha\nabla(\omega^{n+1}+\omega^n)\cdot\nabla\phi\mathrm{d}x\\ &=\Delta t\int_W(\omega^{n+1}-d^{n+1}+\omega^n-d^n)\phi\mathrm{d}x \ , \quad n=N,N-1,\dots,0 \ .\end{aligned} \tag{37.7}$$

To evaluate the right hand side, the area of the well is defined as an subdomain:

```
class WellDomain(SubDomain):
    def inside(self, x, on_boundary):
        return bool((0.2 <= x[0] and x[0] <= 0.3 and \
                     0.2 <= x[1] and x[1] <= 0.3))
```

Next, it gets marked:

```
well = WellDomain()
subdomains = MeshFunction("uint",mesh, mesh.topology().dim())
well.mark(subdomains, 1)
```

An integral over the well area is defined:

```
dxWell = dx(1)
```

The driving source in (37.7) is written as:

```
f = dt*(u1-d1+u0-d0)*phi*dxWell
```

The first line in (37.7) is stated in variational formulation:

```
F = (u_trial-u)*phi*dx \
  + 0.5*dt*( d*dot( grad(u_trial+u), grad(phi) ) )*dx
```

Let DOLFIN sort out left– and right hand sides:

```
a = lhs(F); l = rhs(F)
```

Construct the variational problem:

```
problem = VariationalProblem(a, l+f)
```

And solve it:

```
u = problem.solve()
```

37.7 Numerical experiments

With these preparations, we are now ready to inspect the well output functional (37.2) for possible calibration of the dual lithology model (37.1) to "observed", actually generated synthetic, data. We consider the PDE system (37.1) with discontinuous coefficients

$$\alpha = \begin{cases} \alpha_1 & x \geq 1/2 \\ \alpha_2 & x < 1/2 \end{cases} \quad , \quad \beta = \begin{cases} \beta_1 & x \geq 1/2 \\ \beta_2 & x < 1/2 \end{cases}$$

in the unit square $B = [0,1]^2$. Four wells $W = W_1 \cup W_2 \cup W_3 \cup W_4$ are placed one in each quarter

$$W_4 = [0.3, 0.3] \times [0.7, 0.8] \ , \quad W_3 = [0.7, 0.8] \times [0.7, 0.8] \ ,$$

$$W_1 = [0.2, 0.3] \times [0.2, 0.3] \ , \quad W_2 = [0.7, 0.8] \times [0.2, 0.3] \ .$$

Initially s is constant $s(0,\cdot) = 0.5$ and h is piecewise linear

$$h(0,x,y) = 0.5 \max(\max(0.2,(x-0.1)/2), y - 0.55) \ .$$

The diffusive character of the process is evident from the evolution of h as shown in Figure 37.1. No flow boundary conditions are implemented in all simulations throughout this section.

To inspect the output functional, we generate synthetic data by computing a reference solution. In the first experiment, the reference parameters are $(\alpha_1, \alpha_2) =$

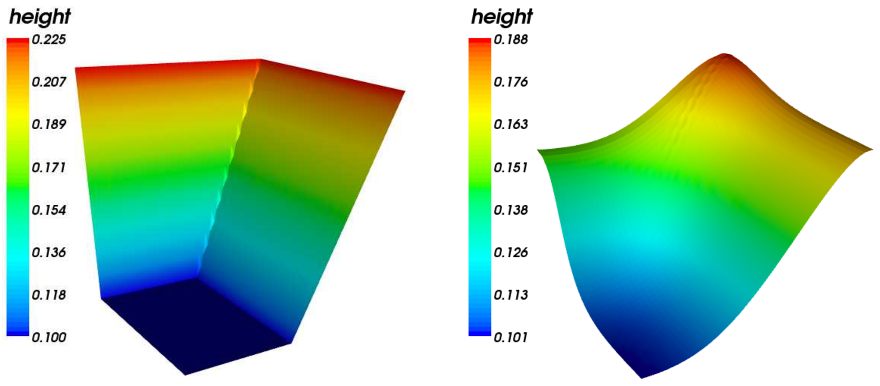


Figure 37.1: Evolution of h, initial left, $t = 0.04$ right.

$(\beta_1, \beta_2) = (0.8, 0.8)$. We fix β to the reference values and scan the well output over the α–range $[0.0, 1.5]^2$. The upper left plot in Figure 37.2 depicts contours of the apparently convex functional, with the reference parameters as the global minimum. Independent Landweber iterations, started in each corner of the domain identify the optimal parameters in typically five steps. The iteration is stopped if $\|\nabla J(p^k)\| \leq 10^{-7}$, an indication that the minimum is reached. The lower left plot shows the corresponding scan over β where $\alpha = (0.8, 0.8)$ is fixed. Obviously the search directions follow the steepest decent, confirming that the gradients are correctly evaluated via the dual solution. In the right column of Figure 37.2 we see results for the same experiments, but with 5% random noise added to the synthetic well data. In this case the optimal parameters are of course not the reference parameters, but still close. The global picture appears stable with respect to noise, suggesting that the concept allows to calibrate diffusive, depositional models to data observed in wells.

Ultimately, the goal is to calibrate all four parameters $\alpha = (\alpha_1, \alpha_2)$ and $\beta = (\beta_1, \beta_2)$ to available data. Figure 37.3 depicts Landweber iterations in four dimensional parameter space. Actually projections onto the α and β coordinate plane are shown. Each subplot depicts one iteration. The initial guess varies from line to line. Obviously, all iterations converge and, without noise added, the reference parameters, $\alpha = \beta = (0.8, 0.8)$, are detected as optimal parameters. Adding 5% random noise to the recorded data, we simulate data observed in wells. In this situation, see the right column, the algorithm identifies optimal parameters, which are clearly off the reference. Fig. 37.5 depicts fifty realizations of this experiments. The distribution of the optimal parameters is shown together with their average in red. The left column in Fig. 37.5 corresponds to the reference parameters $(\alpha_1, \alpha_2) = (\beta_1, \beta_2) = (0.8, 0.8)$ as in Fig. 37.3. The initial guesses vary from row to row and are the same as in Fig. 37.3. On average the calibrated, optimal parameters are close to the reference. typical standard deviations vary from 0.07 to 0.17, depending on the coefficient.

In the next experiments non uniform reference parameters are set for $\alpha = (0.6, 1.0)$ and $\beta = (1.0, 0.6)$. Figure 37.4 shows iterations with the noise–free reference solution used as data on the left hand side. Within the precision of the stopping criterion, the reference parameters are detected. Adding 5% noise to the well data leads to different optimal parameters, just as expected. On average however, the optimal parameters obtained in repeated calibrations match the reference parameters quite well, see Figure 37.5, right hand side.

In the next experiments, β is discontinuous along $y = 1/2$ and piecewise constant in the lower and upper half of the basin

$$\alpha = \begin{cases} \alpha_1 & x \geq 1/2 \\ \alpha_2 & x < 1/2 \end{cases} , \quad \beta = \begin{cases} \beta_1 & y \geq 1/2 \\ \beta_2 & y < 1/2 \end{cases} .$$

In this way the evolution is governed by different diffusion parameters in each quarter of the basin. Having placed one well i each quarter, one can effectively calibrate the model to synthetic data with and without random noise, as shown in Figures 37.6 and 37.7.

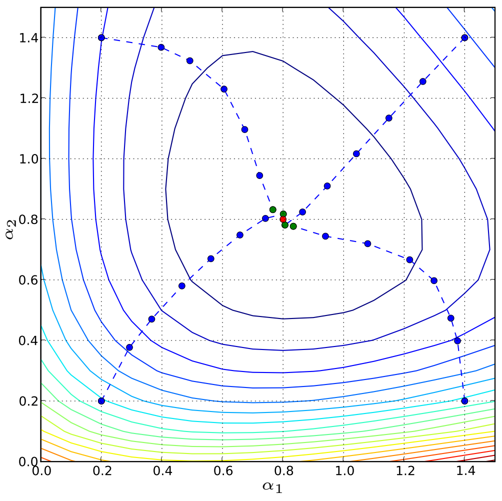
37.8 Results and conclusion

The calibration of piecewise constant diffusion coefficients using local data in a small number of wells is a well behaved inverse problem. The convexity of the output functional, which is the basis for a successful minimization, remains stable with random noise added to the well data.

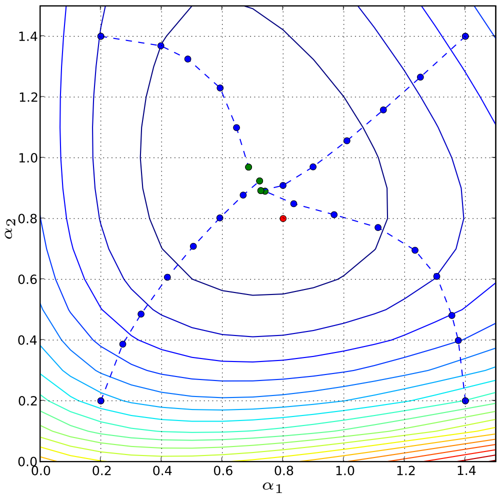
We have automated the calibration of diffusive transport coefficients in two ways: First, the Landweber algorithm, with duality based gradients, automatically detects optimal parameters. Second, the FEniCS project DOLFIN, automatically implements the methods. As the dual problems are derived in variational form, DOLFIN is the appropriate tool for efficient implementation.

Acknowledgments

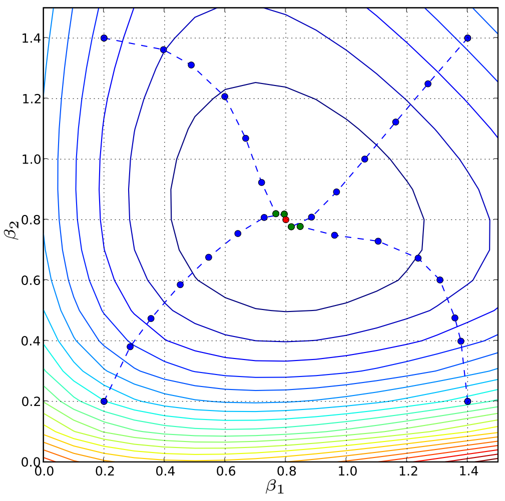
The presented work was funded by a research grant from StatoilHydro. The author wants to thank Are Magnus Bruaset for initiating this work. Many thanks to Bjørn Fredrik Nielsen for thoughtful suggestions regarding inverse problems as well as Anders Logg and Ola Skavhaug for their support regarding the DOLFIN software. The work has been conducted at Simula Research Laboratory as part of CBC, a Center of Excellence awarded by the Norwegian Research Council.



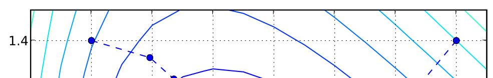
α_2
α_1



α_2
α_1



β_2
β_1



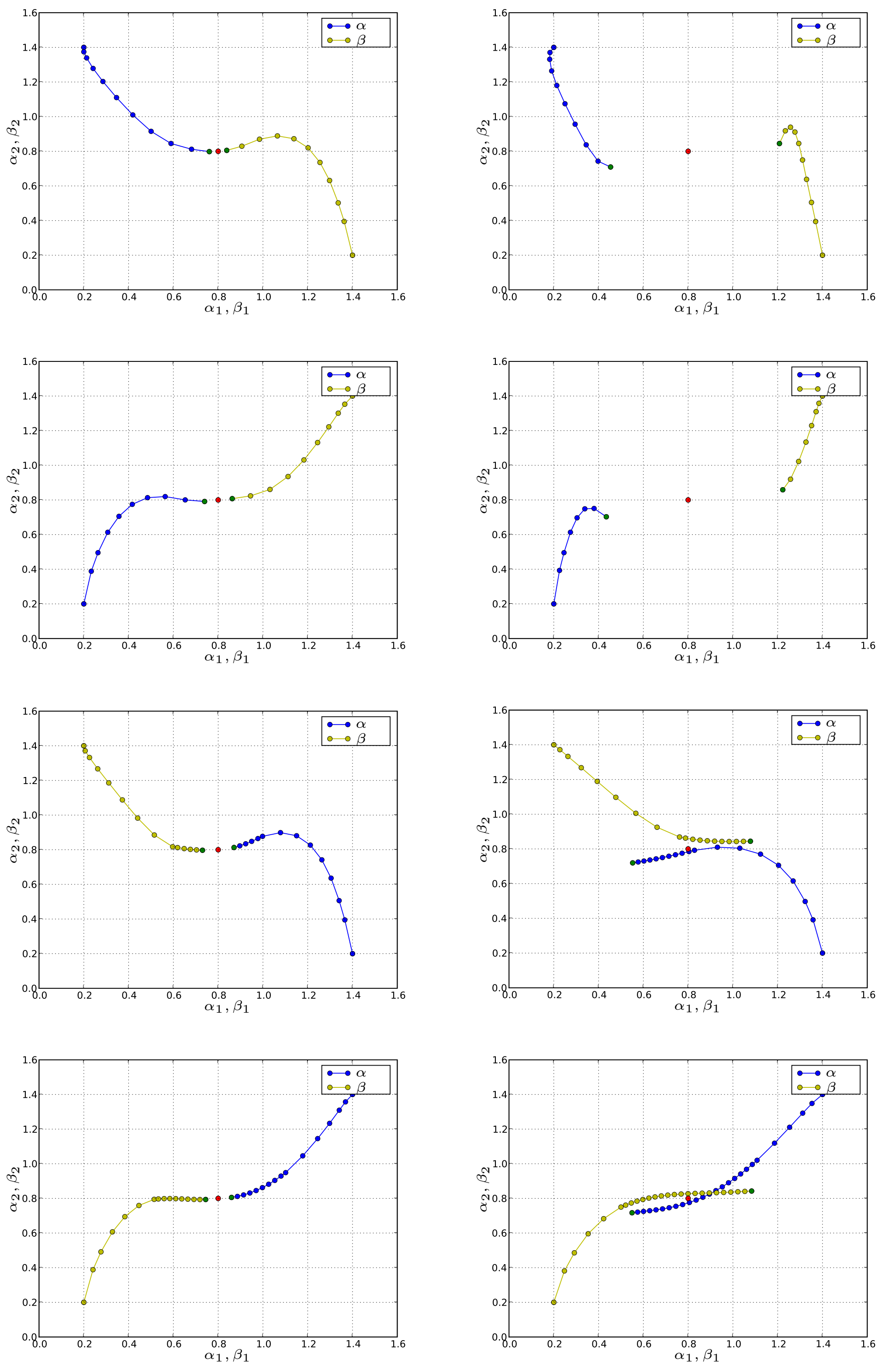


Figure 37.3: Landweber iterations. Clean (left) and noisy data (right).

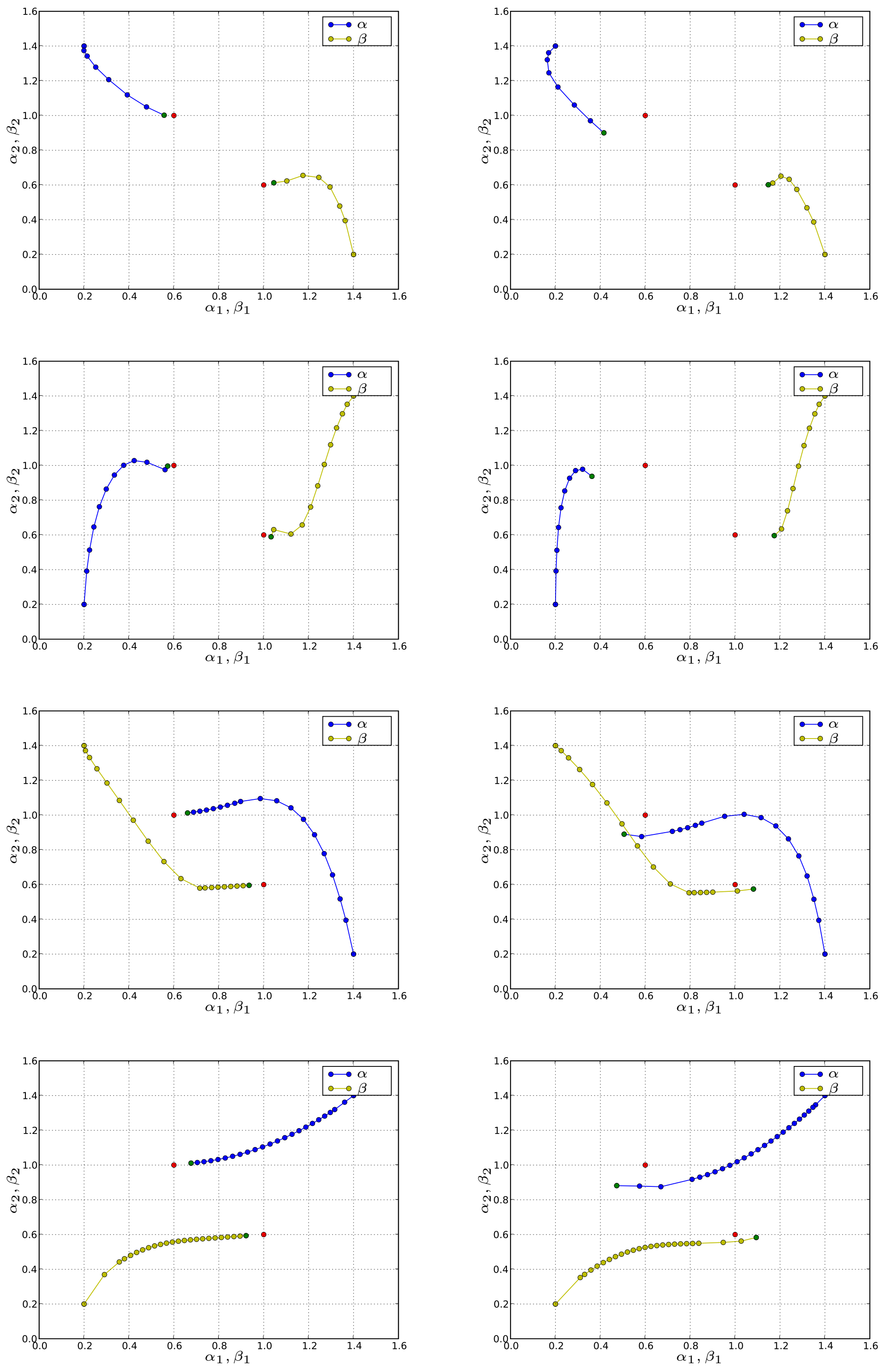


Figure 37.4: Landweber iterations. Clean (left) and noisy data (right).

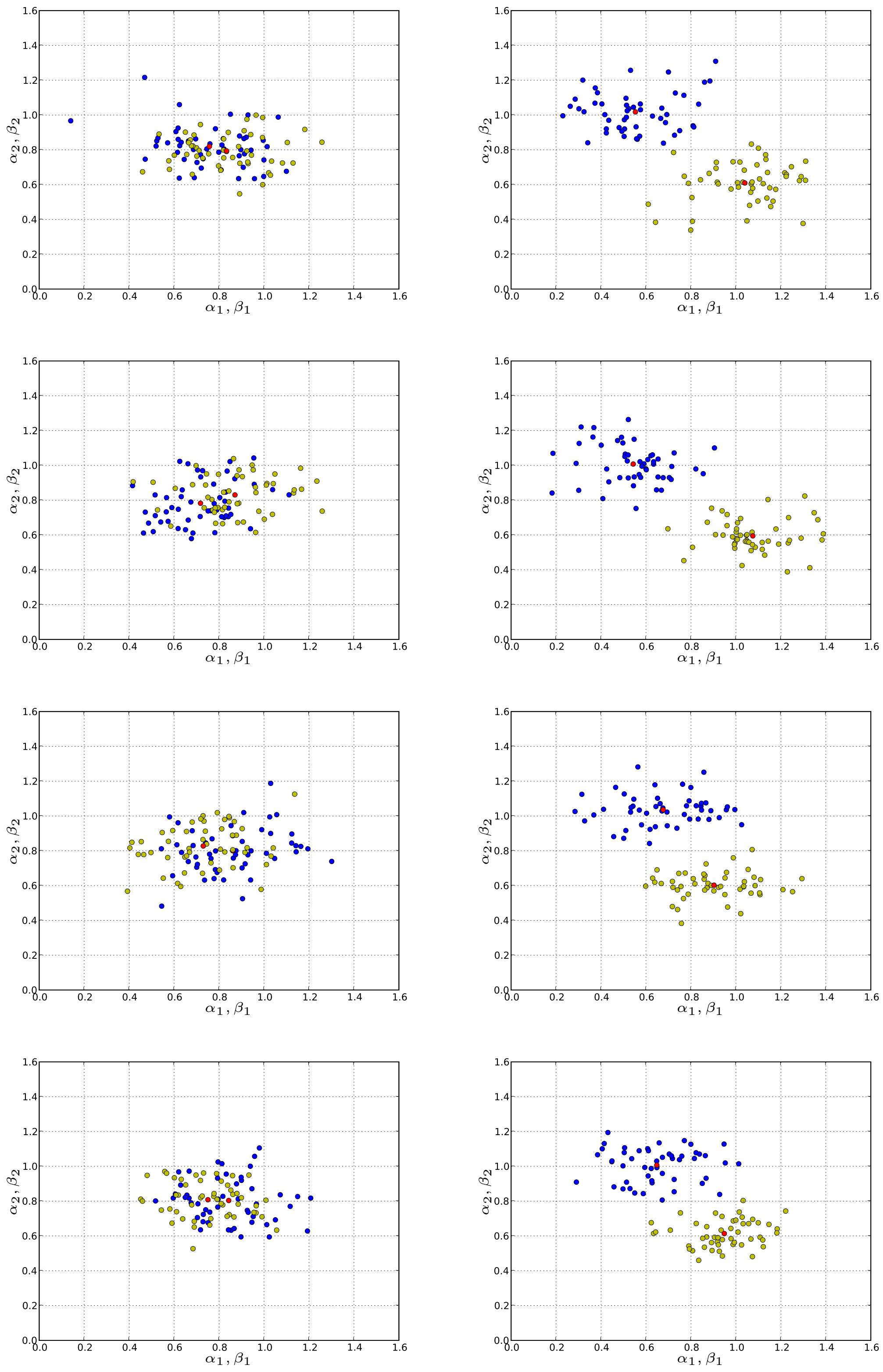


Figure 37.5: Sets of optimal parameters calibrated to noisy data, α blue, β yellow, average red.

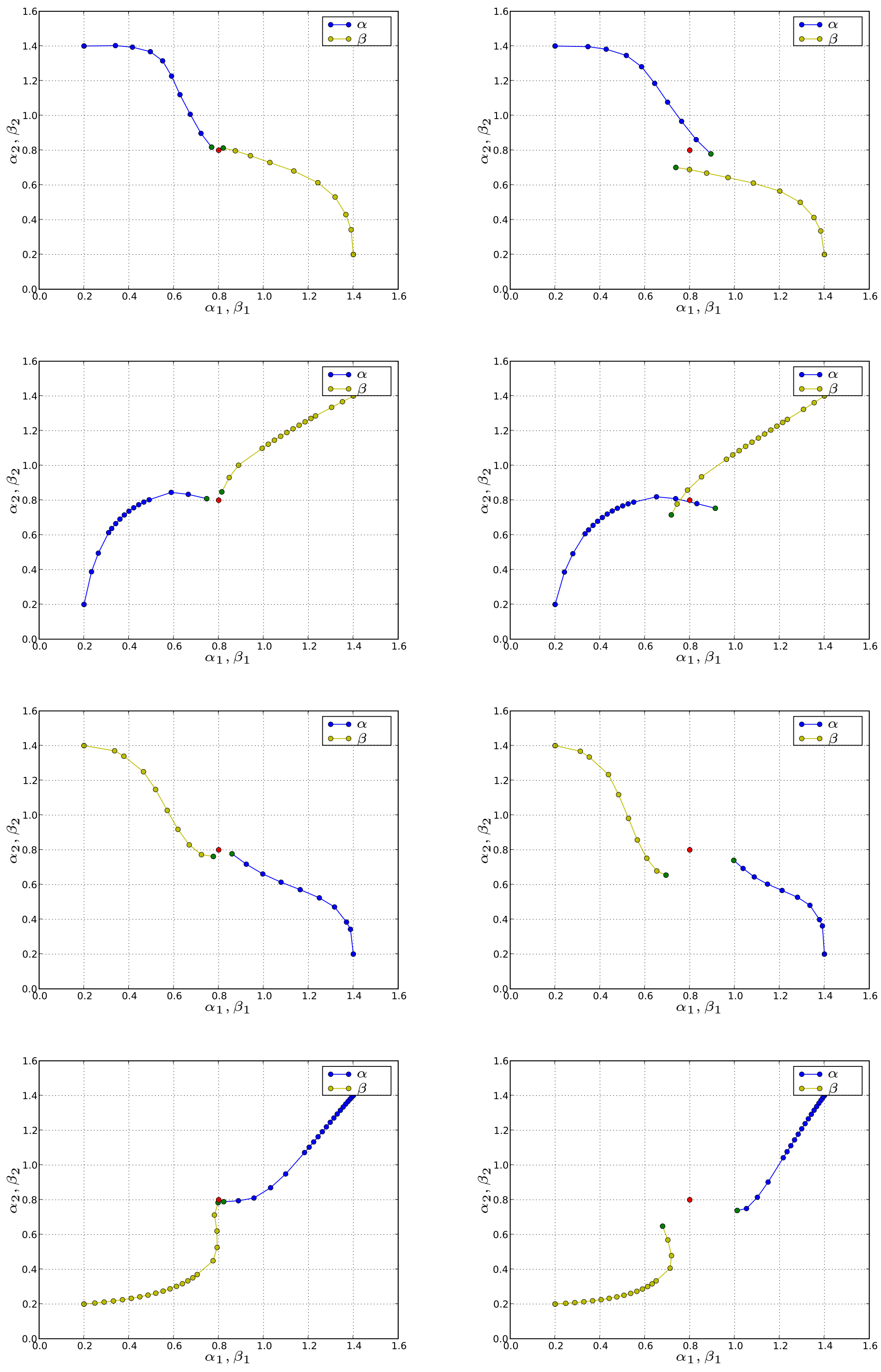


Figure 37.6: Landweber iterations. Clean (left) and noisy data (right).

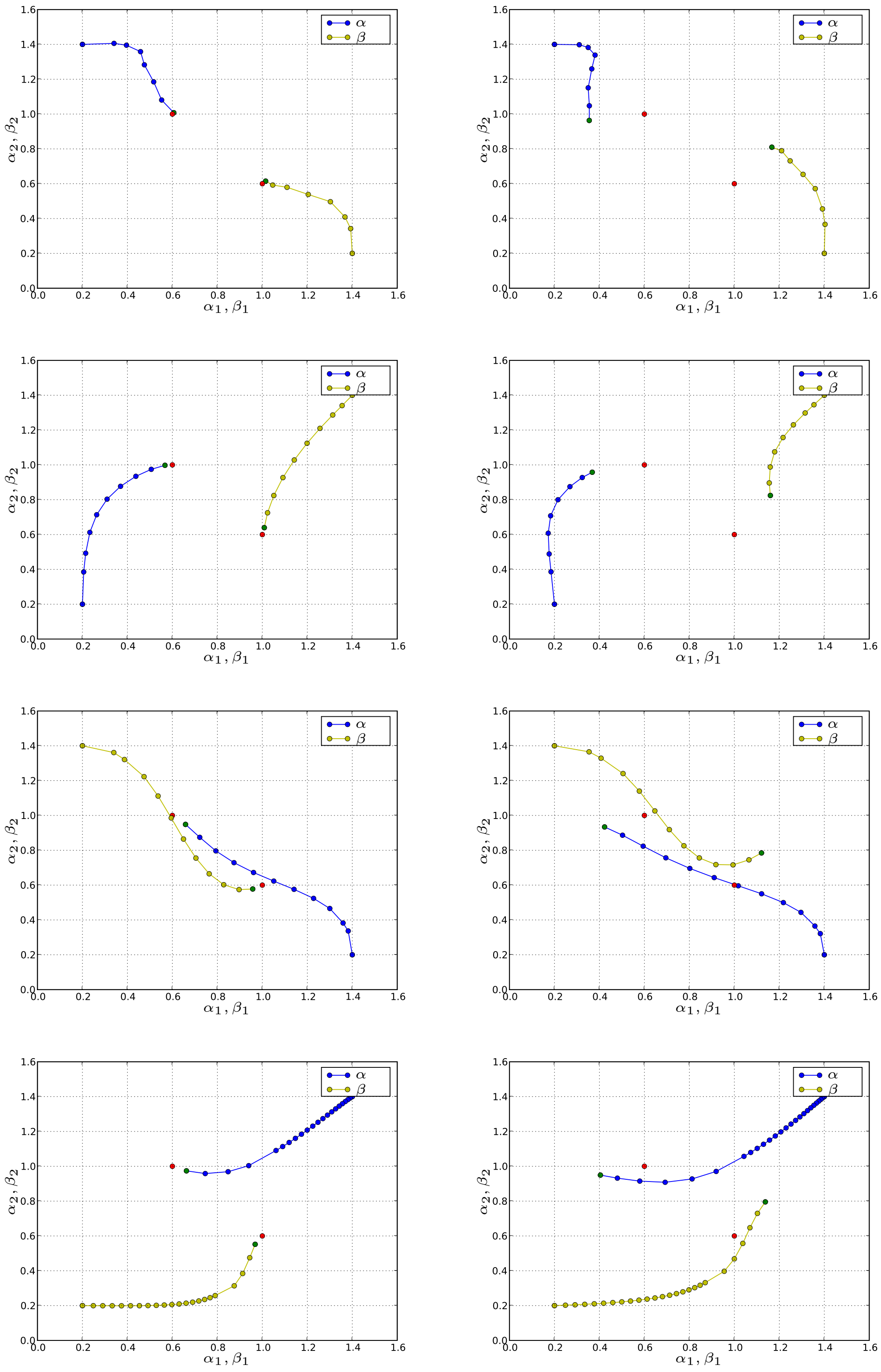


Figure 37.7: Landweber iterations. Clean (left) and noisy data (right).

CHAPTER 38

Computational Compressible Flow and Thermodynamics

By Johan Hoffman, Claes Johnson and Murtazo Nazarov

Chapter ref: **[hoffman-3]**

We test the functionality of FEniCS on the challenge of computational thermodynamics in the form of the EG2 finite element solver of the Euler equations expressing conservation of mass, momentum and energy. We show that EG2 solutions satisfy a 2nd Law formulated in terms of kinetic energy, internal (heat) energy, work and shock/turbulent dissipation, without reference to entropy. We show that the 2nd Law expresses an irreversible transfer of kinetic energy to heat energy in shock/turbulent dissipation arising because the Euler equations lack pointwise solutions. The 2nd Law thus explains the occurence of irreversibility in formally reversible systems as an effect of instability with blow-up of Euler residuals combined with finite precision computation, without resort to statistical mechanics or ad hoc viscous regularization. We simulate the classical Joule or Joule-Thompson experiment of a gas expanding from rest under temperature drop and turbulent dissipation until rest in the double volume.

38.1 FEniCS as Computational Science

The goal of the FEniCS project is to develop software for automated computational solution of differential equations based on a finite element methodology combining generality with efficiency. Thermodynamics is a basic area of continuum mechanics with many important applications, which however is feared by both teachers, students and engineers as being difficult to understand and to apply, principally because of the appearance of turbulence. In this article we show that turbulent thermodynamics can be made understandable and useful by automated computational solution, as a demonstration of the capability of FEniCS.

The biggest mystery of classical thermodynamics is the 2nd Law about entropy and automation cannot harbor any mystery. Expert systems are required for mysteries and FEniCS is not an expert system. Automation requires a continuum mechanics formulation of thermodynamics with a transparent 2nd Law. We present a formulation of thermodynamics based on finite precision computation with a 2nd Law without reference to entropy, which we show can serve as a basis for automated computational simulation of complex turbulent thermodynamics and thus can open to new insight and design, a main goal of FEniCS. In this setting the digital finite element model becomes the real model of the physics of thermodynamics viewed as a form of analog finite precision computation, a model which is open to inspection and analysis because solutions can be computed and put on the table.

38.2 The 1st and 2nd Laws of Thermodynamics

Heat, a quantity which functions to animate, derives from an internal fire located in the left ventricle. (Hippocrates, 460 B.C.)

Thermodynamics is fundamental in a wide range of phenomena from macroscopic to microscopic scales. Thermodynamics essentially concerns the interplay between *heat energy* and *kinetic energy* in a *gas* or *fluid*. Kinetic energy, or *mechanical energy*, may generate heat energy by *compression* or *turbulent dissipation*. Heat energy may generate kinetic energy by *expansion*, but not through a *reverse* process of turbulent dissipation. The industrial society of the 19th century was built on the use of *steam engines*, and the initial motivation to understand thermodynamics came from a need to increase the efficiency of steam engines for conversion of heat energy to useful mechanical energy. Thermodynamics is closely connected to the dynamics of *slightly viscous* and *compressible* gases, since substantial compression and expansion can occur in a gas, but less in fluids (and solids).

The development of classical thermodynamics as a rational science based on logical deduction from a set of axioms, was initiated in the 19th century by Carnot (**?**), Clausius (**?**) and Lord Kelvin (**?**), who formulated the basic axioms in the form of the *1st Law* and the *2nd Law* of thermodynamics. The 1st Law states (for an isolated system) that the *total energy*, the sum of kinetic and heat energy, is conserved. The 1st Law is naturally generalized to include also conservation of mass and Newton's law of conservation of momentum and then can be expressed as the *Euler equations* for a gas/fluid with *vanishing viscosity*.

The 2nd Law has the form of an inequality $dS \geq 0$ for a quantity named *entropy* denoted by S, with dS denoting change thereof, supposedly expressing a basic feature of real thermodynamic processes. The classical 2nd Law states that the entropy cannot decrease; it may stay constant or it may increase, but it can never decrease (for an isolated system).

The role of the 2nd Law is to give a scientific basis to the many observations of *irreversible* processes, that is, processes which cannot be reversed in time, like running a movie backwards. Time reversal of a process with strictly increasing entropy, would correspond to a process with strictly decreasing entropy, which would violate the 2nd Law and therefore could not occur. A perpetum mobile would represent a reversible process and so the role of the 2nd Law is in particular to explain *why* it is impossible to construct a perpetum mobile, and *why* time is moving forward in the direction an *arrow of time*, as expressed by Max Planck (**???**): *Were it not for the existence of irreversible processes, the entire edifice of the 2nd Law would crumble*.

While the 1st Law in the form of the Euler equations expressing conservation of mass, momentum and total energy can be understood and motivated on rational grounds, the nature of the 2nd Law is mysterious. It does not seem to be a consequence of the 1st Law, since the Euler equations seem to be time reversible, and the role of the 2nd Law is to explain irreversibility. Thus questions are lining up: If the 2nd Law is a new independent law of Nature, how can it be justified? What is the physical significance of that quantity named entropy, which Nature can only get more of and never can get rid of, like a steadily accumulating heap of waste? What mechanism prevents Nature from recycling entropy? How can irreversiblity arise in a reversible system? How can viscous dissipation arise in a system with vanishing viscosity? Why is there no *Maxwell demon* (**?**)? Why can a gas by itself expand into a larger volume, but not by itself contract back again, if the motion of the gas molecules is governed by the reversible Newton's laws of motion? Why is there an arrow of time? This article presents answers.

38.3 The Enigma

> Those who have talked of "chance" are the inheritors of antique superstition and ignorance...whose minds have never been illuminated by a ray of scientific thought. (T. H. Huxley)

These were the questions which confronted scientists in the late 19th century, after the introduction of the concept of entropy by Clausius in 1865, and these showed to be tough questions to answer. After much struggle, agony and debate, the agreement of the physics community has become to view *statistical mechanics* based on an assumption of *molecular chaos* as developed by Boltzmann (**?**), to offer a rationalization of the classical 2nd Law in the form of a tendency of (isolated) physical processes to move from improbable towards more probable states, or from ordered to less ordered states. Boltzmann's assumption of molecular chaos in a dilute gas of colliding molecules, is that two molecules about to collide have independent velocities, which led to the *H-theorem* for *Boltzmann's equations* stating that a certain quantity denoted by H could not decrease and thus could serve as an entropy defining an arrow of time. Increasing disorder would thus represent increasing entropy, and the classical 2nd Law would reflect the eternal pessimisticts idea that things always get more messy, and that there is really no limit to this, except when everything is as messy as it can ever get. Of course, experience could give (some) support this idea, but the trouble is that it prevents things from ever becoming less messy or more structured, and thus may seem a bit too pessimistic. No doubt, it would seem to contradict the many observations of *emergence* of ordered non-organic structures (like crystals or waves and cyclones) and organic structures (like DNA and human beings), seemingly out of disordered chaos, as evidenced by the physics Nobel Laureate Robert Laughlin.

Most trained thermodynamicists would here say that emergence of order out of chaos, in fact does not contradict the classical 2nd Law, because it concerns "non-isolated systems". But they would probably insist that the Universe as a whole (isolated system) would steadily evolve towards a "heat-death" with maximal entropy/disorder (and no life), thus fulfilling the pessimists expectation. The question from where the initial order came from, would however be left open.

The standard presentation of thermodynamics based on the 1st and 2nd Laws, thus involves a mixture of deterministic models (Boltzmann's equations with the H-theorem) based on statistical assumptions (molecular chaos) making the subject admittedly difficult to both learn, teach and apply, despite its strong importance. This is primarily because the question *why* necessarily $dS \geq 0$ and never $dS < 0$, is not given a convincing understandable answer. In fact, statistical mechanics allows $dS < 0$, although it is claimed to be very unlikely. The basic objective of statistical mechanics as the basis of classical thermodynamics, thus is to (i) give the entropy a physical meaning, and (ii) to motivate its tendency to (usually) increase. Before statistical mechanics, the 2nd Law was viewed as an experimental fact, which could not be rationalized theoretically. The classical view on the 2nd Law is thus either as a statistical law of large numbers or as a an experimental fact, both without a rational deterministic mechanistic theoretical foundation. The problem with thermodynamics in this form is that it is understood by very few, if any:

- *Every mathematician knows it is impossible to understand an elementary course in thermodynamics.* (V. Arnold)
- *...no one knows what entropy is, so if you in a debate use this concept, you will always have an advantage.* (Von Neumann to Shannon)
- *As anyone who has taken a course in thermodynamics is well aware, the mathematics used in proving Clausius' theorem (the 2nd Law) is of a very special kind, having only the most tenuous relation to that known to mathematicians.* (**?**)

- *Where does irreversibility come from? It does not come form Newton's laws. Obviously there must be some law, some obscure but fundamental equation. perhaps in electricity, maybe in neutrino physics, in which it does matter which way time goes.* (**?**)
- *For three hundred years science has been dominated by a Newtonian paradigm presenting the World either as a sterile mechanical clock or in a state of degeneration and increasing disorder...It has always seemed paradoxical that a theory based on Newtonian mechanics can lead to chaos just because the number of particles is large, and it is subjectively decided that their precise motion cannot be observed by humans... In the Newtonian world of necessity, there is no arrow of time. Boltzmann found an arrow hidden in Nature's molecular game of roulette.* (Paul Davies)
- *The goal of deriving the law of entropy increase from statistical mechanics has so far eluded the deepest thinkers.* (**?**)
- *There are great physicists who have not understood it.* (Einstein about Boltzmann's statistical mechanics)

38.4 Computational Foundation

In this note we present a foundation of thermodynamics, further elaborated in (Nazarov, 2009, **?**), where the basic assumption of statistical mechanics of molecular chaos, is replaced by *deterministic finite precision computation*, more precisely by a *least squares stabilized finite element method* for the Euler equations, referred to as *Euler General Galerkin* or *EG2*. We thus view EG2 as the physical model of thermodynamics, that is the Euler equations together with a computational solution procedure, and not just the Euler equations without constructive solution procedure as in a classical non-computational approach.

Using EG2 as a model of thermodynamics changes the questions and answers and opens new possibilities of progress together with new challenges to mathematical analysis and computation. The basic new feature is that EG2 solutions are computed and thus are available to inspection. This means that the analysis of solutions shifts from *a priori* to *a posteriori*; after the solution has been computed it can be inspected.

Inspecting computed EG2 solutions we find that they are *turbulent* and have *shocks*, which is identified by pointwise large Euler residuals, reflecting that pointwise solutions to the Euler equations are lacking. The enigma of thermodynamics is thus the enigma of turbulence (since the basic nature of shocks is understood). Computational thermodynamics thus essentially concerns computational turbulence. In this note and (**?**) we present evidence that EG2 opens to a resolution of the enigma of turbulence and thus of thermodynamics.

The fundamental question concerns *wellposedness* in the sense of Hadamard, that is what aspects or *outputs* of turbulent/shock solutions are stable under perturbations in the sense that small perturbations have small effects. We show that wellposedness of EG2 solutions can be tested a posteriori by computationally solving a *dual linearized problem*, through which the output sensitivity of non-zero Euler residuals can be estimated. We find that mean-value outputs such as drag and lift and total turbulent dissipation are wellposed, while point-values of turbulent flow are not. We can thus a posteriori in a case by case manner, assess the quality of EG2 solutions as solutions of the Euler equations.

We formulate a *2nd Law* for EG2 without the concept of entropy, in terms of the basic physical quantities of kinetic energy K, heat energy E, rate of *work* W and shock/turbulent dissipation $D > 0$. The new 2nd Law reads

$$\dot{K} = W - D, \quad \dot{E} = -W + D, \tag{38.1}$$

where the dot indicates time differentiation. Slightly viscous flow always develops turbulence/shocks with $D > 0$, and the 2nd Law thus expresses an irreversible transfer of kinetic energy into heat energy, while the total energy $E + K$ remains constant.

With the 2nd Law in the form (38.1), we avoid the (difficult) main task of statistical mechanics of specifying the physical significance of entropy and motivating its tendency to increase by probabilistic considerations based on (tricky) combinatorics. Thus using *Ockham's razor* we rationalize a scientific theory of major importance making it both more understandable and more useful. The new 2nd Law is closer to classical Newtonian mechanics than the 2nd Law of statistical mechanics, and thus can be viewed to be more fundamental.

The new 2nd Law is a consequence of the 1st Law in the form of the Euler equations combined with EG2 finite precision computation effectively introducing viscosity and viscous dissipation. These effects appear as a consequence of the non-existence of pointwise solutions to the Euler equations reflecting instabilities leading to the development shocks and turbulence in which large scale kinetic energy is transferred to small scale kinetic energy in the form of heat energy. The viscous dissipation can be interpreted as a penalty on pointwise large Euler residuals arising in shocks/turbulence, with the penalty being directly coupled to the violation. EG2 thus explains the 2nd Law as a consequence of the non-existence of pointwise solutions with small Euler residuals. This offers an understanding to the emergence of irreversible solutions of the formally reversible Euler equations. If pointwise solutions had existed, they would have been reversible without dissipation, but they don't exist, and the existing computational solutions have dissipation and thus are irreversible.

38.5 Viscosity Solutions

An EG2 solution can be viewed as particular *viscosity solution* of the Euler equations, which is a solution of *regularized Euler equations* augmented by additive terms modeling viscosity effects with small viscosity coefficients. The effective viscosity in an EG2 solution typically may be comparable to the mesh size.

For incompressible flow the existence of viscosity solutions, with suitable solution dependent viscosity coefficients, can be proved a priori using standard techniques of analytical mathematics. Viscosity solutions are pointwise solutions of the regularized equations. But already the most basic problem with constant viscosity, the incompressible Navier-Stokes equations for a Newtonian fluid, presents technical difficulties, and is one of the open Clay Millennium Problems.

For compressible flow the technical complications are even more severe, and it is not clear which viscosities would be required for an analytical proof of the existence of viscosity solutions (**?**) to the Euler equations. Furthermore, the question of wellposedness is typically left out, as in the formulation of the Navier-Stokes Millennium Problem, with the motivation that first the existence problem has to be settled. Altogether, analytical mathematics seems to have little to offer a priori concerning the existence and wellposedness of solutions of the compressible Euler equations. In contrast, EG2 computational solutions of the Euler equations seem to offer a wealth of information a posteriori, in particular concerning wellposedness by duality.

An EG2 solution thus can be viewed as a specific viscosity solution with a specific regularization from the least squares stabilization, in particular of the momentum equation, which is necessary because pointwise momentum balance is impossible to achieve in the presence of shocks/turbulence. The EG2 viscosity can be viewed to be the minimal viscosity required to handle the contradiction behind the non-existence of pointwise solutions. For a shock EG2 could then be directly interpreted as a certain physical mechanism preventing a shock wave from turning over, and for turbulence as a form of automatic computational turbulence model.

EG2 thermodynamics can be viewed as form of deterministic chaos, where the mechanism is open to inspection and can be used for prediction. On the other hand, the mechanism of statistical mechanics is not open to inspection and can only be based on ad hoc assumption. If Boltzmann's assumption of molecular chaos cannot be justified, and is not needed, why consider it at all, (**?**)?

38.6 Joule's 1845 Experiment

To illustrate basic aspects of thermodynamics, we recall Joule's experiment from 1845 with a gas initially at rest, or in equilibrium, at a certain temperature and density in a certain volume immersed into a container of water, see Fig. 38.1. At initial time a valve was opened and the gas was allowed to expand into the double volume while the temperature change in the water was carefully measured by Joule. To the great surprise of both Joule and the scientific community, no

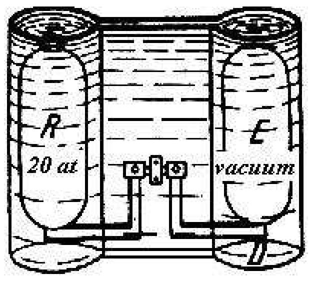


Fig. 358 Concerning overflowing experiment of Joule (Scientific Papers). R contains at first air compressed to 20 atm, E is initially a vacuum, D the tube

Figure 38.1: Joule's 1845 experiment

change of the temperature of the water could be detected, in contradiction with the expectation that the gas would cool off under expansion. Moreover, the expansion was impossible to reverse; the gas had no inclination to contract back to the original volume.

We simulate Joule's experiment computationally using EG2: At initial time a valve is opened in a channel connecting two cubical chambers, a left and a right chamber, filled with gas of the same temperature but different density/pressure with high density/pressure in the left and low in the right chamber. Figs. 38.2 - 38.4 display the time-evolution of mean temperature, density, kinetic energy, pressure and turbulent dissipation in the left and right chambers. Figs. 38.5 - 38.7 give snapshots of the distribution of temperature and speed at an intermediate time.

We see that temperature drops in the left chamber as the gas expands with heat energy transforming to kinetic energy with a maximal temperature drop in the channel. The cool expanding gas is heated in the right chamber by compression and shock/turbulence dissipation. The mean temperature thus drops in the left chamber and increases in the right and after a slight rebounce settles to a remaining density/temperature gap as the gas comes to rest with the same pressure in the left and right chambers and the same total heat energy as before expansion.

From the 1st Law alone there are many different possible end states with varying gaps in density/temperature. It is the 2nd Law which determines the size of the gap, which relates to the amount of turbulent/shock dissipation in the left and right chambers, which is determined by the dynamics of the process including the distribution of turbulence/shock dissipation.

Classical thermodynamics focusing on equilibrium states does not tell which from a range of possible equilibrium end states with varying gaps, will actually be realized, because the true end state depends on the dynamics of the process.

If anything, classical thermodynamics would predict an end state with zero gap (constant enthalpy), which we have seen is incorrect. In short, classical equilibrium thermodynamics excluding dynamics cannot correctly predict equilibrium end states, and thus has little practical value.

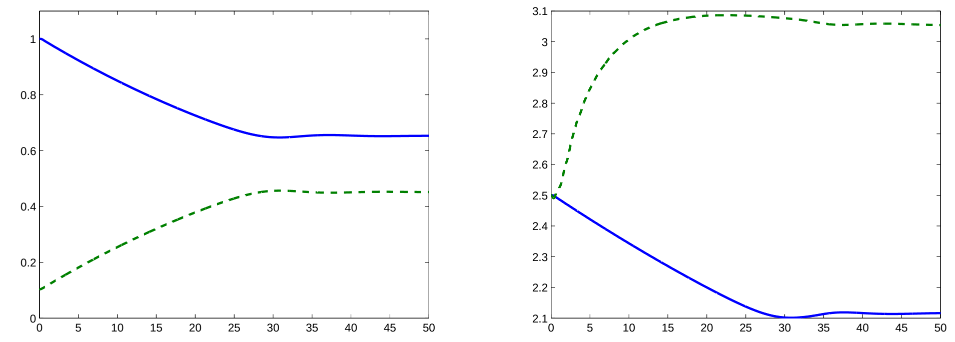


Figure 38.2: Evolution in time of mean density and temperature in left and right chambers

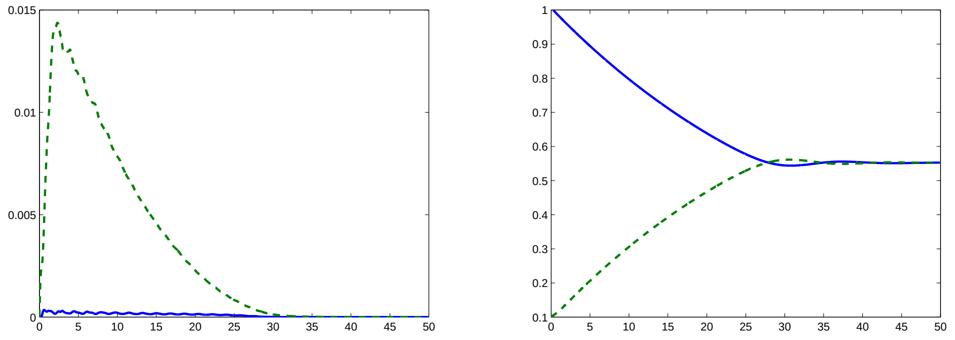


Figure 38.3: Evolution in time of mean kinetic energy and pressure in left and right chambers

The 2nd Law states that reversal of the process with the gas contracting back to the original small volume, is impossible because the only way the gas can be put into motion without external forcing is by expansion: self-expansion is possible, but not self-constraction.

We are thus able to analyze and understand the dynamics of the Joule experiment using the 1st and the new form of the 2nd law. The experiment displays the expansion phase of a compression refrigerator with heat being moved by expansion from the left chamber in contact with the inside of the refrigerator, into the right chamber in contact with the outside. The cycle is closed by recompression under outside cooling. The efficiency connects to the temperature drop in the left chamber and the gap, with efficiency suffering from rebounce to a small gap.

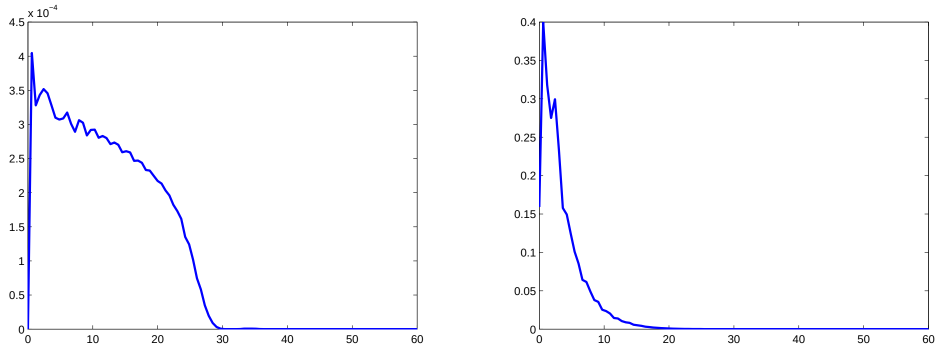


Figure 38.4: Evolution in time of mean turbulent dissipation in left and right chambers

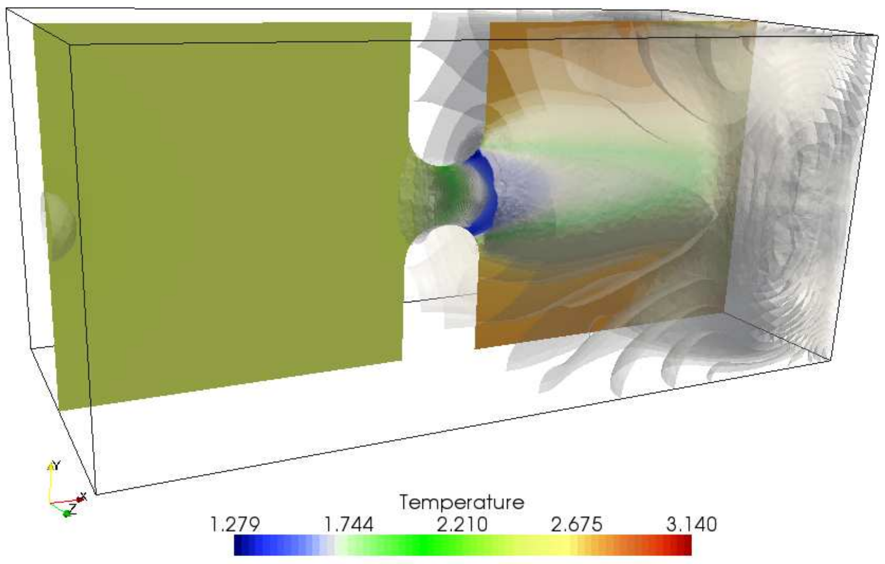


Figure 38.5: Distribution of gas temperature at $T = 3$

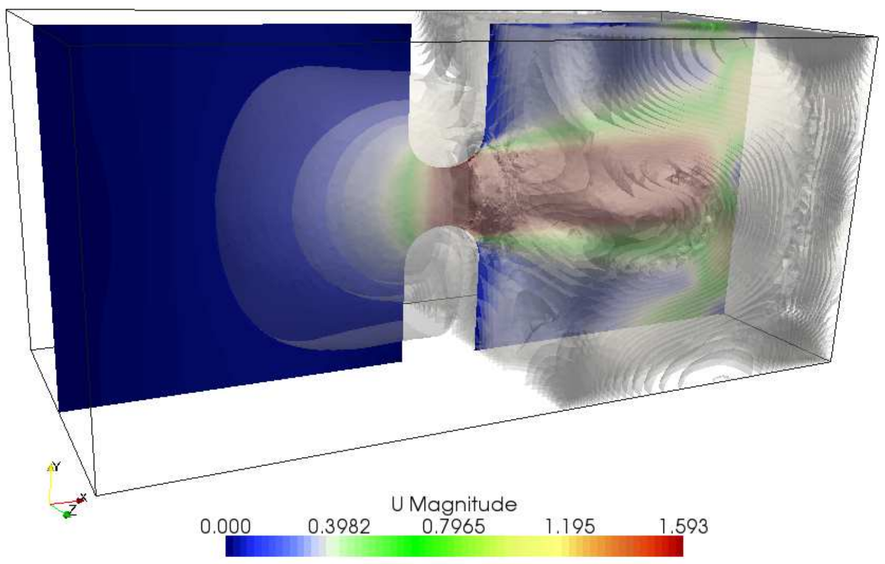


Figure 38.6: Distribution of gas speed at $T = 3$

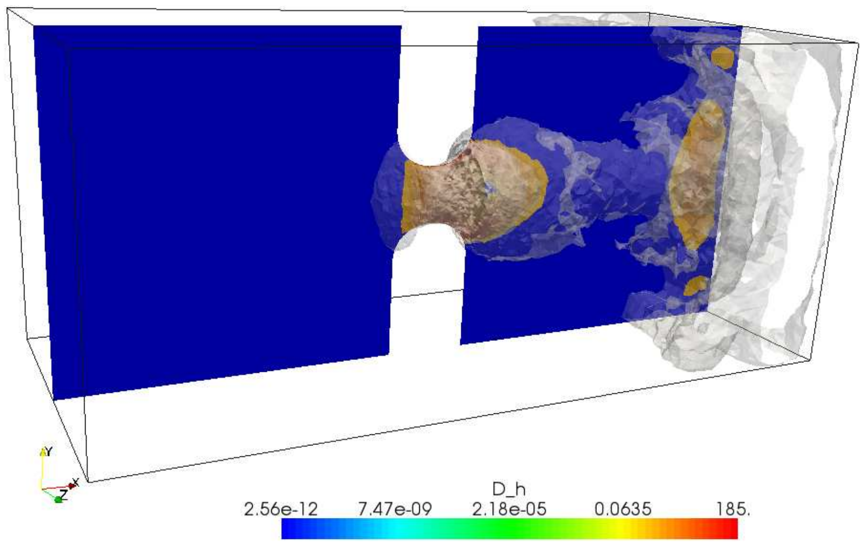


Figure 38.7: Distribution of turbulent dissipation at $T = 3$

38.7 The Euler Equations

We consider the Euler equations for an inviscid perfect gas enclosed in a volume Ω in $\mathbb{R}^3$ with boundary Γ over a time interval $I = (0, 1]$ expressing conservation of *mass density* ρ, *momentum* $m = (m_1, m_2, m_3)$ and *internal energy* e: Find $\hat{u} = (\rho, m, e)$ depending on $(x, t) \in Q \equiv \Omega \times I$ such that

$$\begin{array}{rcl} R_\rho(\hat{u}) \equiv \dot{\rho} + \nabla \cdot (\rho u) & = & 0 \quad \text{in } Q, \\ R_m(\hat{u}) \equiv \dot{m} + \nabla \cdot (mu + p) & = & f \quad \text{in } Q, \\ R_e(\hat{u}) \equiv \dot{e} + \nabla \cdot (eu) + p\nabla \cdot u & = & g \quad \text{in } Q, \\ u \cdot n & = & 0 \quad \text{on } \Gamma \times I \\ \hat{u}(\cdot, 0) & = & \hat{u}^0 \quad \text{in } \Omega, \end{array} \tag{38.2}$$

where $u = \frac{m}{\rho}$ is the velocity, $p = (\gamma - 1)e$ with $\gamma > 1$ a *gas constant*, f is a given volume force, g a heat source/sink and $\hat{u}^0$ a given initial state. We here express energy conservation in terms of the internal energy $e = \rho T$, with T the temperature, and not as conservation of the *total energy* $\epsilon = e + k$ with $k = \frac{\rho v^2}{2}$ the *kinetic energy*, in the form $\dot{\epsilon} + \nabla \cdot (\epsilon u) = 0$. Because of the appearance of shocks/turbulence, the Euler equations lack pointwise solutions, except possible for short time, and regularization is therefore necessary. For a mono-atomic gas $\gamma = 5/3$ and (38.2) then is a *parameter-free model*, the ideal form of mathematical model according to Einstein..

38.8 Energy Estimates for Viscosity Solutions

For the discussion we consider the following regularized version of (38.2) assuming for simplicity that $f = 0$ and $g = 0$: Find $\hat{u}_{\nu,\mu} \equiv \hat{u} = (\rho, m, e)$ such that

$$\begin{array}{rcl} R_\rho(\hat{u}) & = & 0 \quad \text{in } Q, \\ R_m(\hat{u}) & = & -\nabla \cdot (\nu \nabla u) + \nabla(\mu p \nabla \cdot u) \quad \text{in } Q, \\ R_e(\hat{u}) & = & \nu |\nabla u|^2 \quad \text{in } Q, \\ u & = & 0 \quad \text{on } \Gamma \times I, \\ \hat{u}(\cdot, 0) & = & \hat{u}^0 \quad \text{in } \Omega, \end{array} \tag{38.3}$$

where $\nu > 0$ is a *shear viscosity* $\mu >> \nu \geq 0$ if $\nabla \cdot u > 0$ in expansion (with $\mu = 0$ if $\nabla \cdot u \leq 0$ in compression), is a small *bulk viscosity*, and we use the notation $|\nabla u|^2 = \sum_i |\nabla u_i|^2$. We shall see that the bulk viscosity is a safety feature putting a limit to the work $p\nabla \cdot u$ in expansion appearing in the energy balance.

We note that only the momentum equation is subject to viscous regularization. Further, we note that the shear viscosity term in the momentum equation multiplied by the velocity u (and formally integrated by parts) appears as a positive right hand side in the equation for the internal energy, reflecting that the dissipation from shear viscosity is transformed into internal heat energy. In contrast, the dissipation from the bulk viscosity represents another form of internal energy not accounted for as heat energy, acting only as a safety feature in the sense that its contribution to the energy balance in general will be small, while that from the shear viscosity in general will be substantial reflecting shock/turbulent dissipation.

Below we will consider instead regularization by EG2 with the advantage that the EG2 solution is computed and thus is available to inspection, while $\hat{u}_{\nu,\mu}$ is not. We shall see that EG2 regularization can be interpreted as a (mesh-dependent) combination of bulk and shear viscosity and thus (38.3) can be viewed as an analytical model of EG2 open to simple form of analysis in the form of energy estimates.

As indicated, the existence of a pointwise solution $\hat{u} = \hat{u}_{\nu,\mu}$ to the regularized equations (38.3) is an open problem of analytical mathematics, although with suitable additional regularization it could be possible to settle (**?**). Fortunately, we can leave this problem aside, since EG2 solutions will be shown to exist a posteriori by computation. We thus formally assume that (38.3) admits a pointwise solution, and derive basic energy estimates which will be paralleled below for EG2. We thus use the regularized problem (38.3) to illustrate basic features of EG2, including the 2nd Law.

We shall prove now that a regularized solution $\hat{u}$ is an approximate solution of the Euler equations in the sense that $R_\rho(\hat{u}) = 0$ and $R_e(\hat{u}) \geq 0$ pointwise, $R_m(\hat{u})$ is weakly small in the sense that

$$\|R_m(\hat{u})\|_{-1} \leq \frac{\sqrt{\nu}}{\sqrt{\mu}} + \sqrt{\mu} << 1, \tag{38.4}$$

where $\|\cdot\|_{-1}$ denotes the $L_2(I; H^{-1}(\Omega))$-norm, and the following 2nd Law holds:

$$\dot{K} \leq W - D, \quad \dot{E} = -W + D, \tag{38.5}$$

where

$$K = \int_\Omega k\,dx, \quad E = \int_\Omega e\,dx, \quad W = \int_\Omega p\nabla\cdot u\,dx, \quad D = \int_\Omega \nu|\nabla u|^2\,dx.$$

Choosing $\nu << \mu$ we can assure that $\|R_m(\hat{u}_{\nu,\mu})\|_{-1}$ is small. We can view the 2nd Law as a compensation for the fact that the momentum equation is only satisfied in a weak sense, and the equation for internal energy with inequality.

The 2nd Law (38.5) states an irreversible transfer of kinetic energy to heat energy in the presence of shocks/turbulence with $D > 0$, which is the generic case. On the other hand, the sign of W is variable and thus the corresponding energy transfer may go in either direction.

The basic technical step is to multiply the momentum equation by u, and use the mass balance equation in the form $\frac{|u|^2}{2}(\dot{\rho} + \nabla\cdot(\rho u)) = 0$, to get

$$\dot{k} + \nabla\cdot(ku) + p\nabla\cdot u - \nabla(\mu p\nabla\cdot u)\cdot u - \nabla\cdot(\nu\nabla u)\cdot u = 0. \tag{38.6}$$

By integration in space it follows that $\dot{K} \leq W - D$, and similarly it follows that $\dot{E} = -W + D$ from the equation for e, which proves the 2nd Law. Adding next (38.6) to the equation for the internal energy e and integrating in space, gives

$$\dot{K} + \dot{E} + \int_\Omega \mu p (\nabla \cdot u)^2 \, dx = 0,$$

and thus after integration in time

$$K(1) + E(1) + \int_Q \mu p (\nabla \cdot u)^2 \, dxdt = K(0) + E(0). \tag{38.7}$$

We now need to show that $E(1) \geq 0$ (or more generally that $E(t) > 0$ for $t \in I$), and to this end we rewrite the equation for the internal energy as follows:

$$D_u e + \gamma e \nabla \cdot u = \nu |\nabla u|^2,$$

where $D_u e = \dot{e} + u \cdot \nabla e$ is the material derivative of e following the fluid particles with velocity u. Assuming that $e(x, 0) > 0$ for $x \in \Omega$, it follows that $e(x, 1) > 0$ for $x \in \Omega$, and thus $E(1) > 0$. Assuming $K(0) + E(0) = 1$ the energy estimate (38.7) thus shows that

$$\int_Q \mu p (\nabla \cdot u)^2 \, dxdt \leq 1, \tag{38.8}$$

and also that $E(t) \leq 1$ for $t \in I$. Next, integrating (38.6) in space and time gives, assuming for simplicity that $K(0) = 0$,

$$K(1) + \int_Q \nu (\Delta u)^2 dxdt = \int_Q p \nabla \cdot u dxdt - \int_Q \mu p (\nabla \cdot u)^2 dxdt \leq \frac{1}{\mu} \int_Q p dxdt \leq \frac{1}{\mu},$$

where we used that $\int_Q p dxdt = (\gamma - 1) \int_Q e dxdt \leq \int_I E(t) dt \leq 1$. It follows that

$$\int_Q \nu |\nabla u|^2 dxdt \leq \frac{1}{\mu}. \tag{38.9}$$

By standard estimation (assuming that p is bounded), it follows from (38.8) and (38.9) that

$$\|R_m(\hat{u})\|_{-1} \leq C(\sqrt{\mu} + \frac{\sqrt{\nu}}{\sqrt{\mu}}),$$

with C a constant of moderate size, which completes the proof. As indicated, $\|R_m(\hat{u})\|_{-1}$ is estimated by computation, as shown below. The role of the analysis is thus to rationalize computational experience, not to replace it.

38.9 Compression and Expansion

The 2nd Law (38.5) states that there is a transfer of kinetic energy to heat energy if $W < 0$, that is under compression with $\nabla \cdot u < 0$, and a transfer from heat to kinetic energy if $W > 0$, that is under expansion with $\nabla \cdot u > 0$. Returning to Joule's experiment, we see by the 2nd Law that contraction back to the original volume from the final rest state in the double volume, is impossible, because the only way the gas can be set into motion is by expansion. To see this no reference to entropy is needed.

38.10 A 2nd Law without Entropy

We note that the 2nd Law (38.5) is expressed in terms of the physical quantities of kinetic energy K, heat energy E, work W, and dissipation D and does not involve any concept of entropy. This relieves us from the task of finding a physical significance of entropy and justification of a classical 2nd Law stating that entropy cannot decrease. We thus circumvent the main difficulty of classical thermodynamics based on statistical mechanics, while we reach the same goal as statistical mechanics of explaining irreversibility in formally reversible Newtonian mechanics.

We thus resolve *Loschmidt's paradox* (**?**) asking how irreversibility can occur in a formally reversible system, which Boltzmann attempted to solve. But Loschmidt pointed out that Boltzmann's equations are not formally reversible, because of the assumption of molecular chaos that velocities are independent before collision, and thus Boltzmann effectively assumes what is to be proved. Boltzmann and Loschmidt's met in heated debates without conclusion, but after Boltzmann's tragic death followed by the experimental verification of the molecular nature of gases, Loschmidt's paradox evaporated as if it had been resolved, while it had not. Postulating molecular chaos still amounts to assume what is to be proved.

38.11 Comparison with Classical Thermodynamics

Classical thermodynamics is based on the relation

$$Tds = dT + pdv, \tag{38.10}$$

where ds represents change of entropy s per unit mass, dv change of volume and dT denotes the change of temperature T per unit mass, combined with a 2nd Law in the form $ds \geq 0$. On the other hand, the new 2nd Law takes the symbolic form

$$dT + pdv \geq 0, \tag{38.11}$$

effectively expressing that $Tds \geq 0$, which is the same as $ds \geq 0$ since $T > 0$. In symbolic form the new 2nd Law thus expresses the same as the classical 2nd Law, without referring to entropy.

Integrating the classical 2nd Law (38.10) for a perfect gas with $p = (\gamma - 1)\rho T$ and $dv = d(\frac{1}{\rho}) = -\frac{d\rho}{\rho^2}$, we get

$$ds = \frac{dT}{T} + \frac{p}{T}d(\frac{1}{\rho}) = \frac{dT}{T} + (1-\gamma)\frac{d\rho}{\rho},$$

and we conclude that with $e = \rho T$,

$$s = \log(T\rho^{1-\gamma}) = \log(\frac{e}{\rho^\gamma}) = \log(e) - \gamma\log(\rho) \qquad \textbf{(38.12)}$$

up to a constant. Thus, the entropy $s = s(\rho, e)$ for a perfect gas is a function of the physical quantities ρ and $e = \rho T$, thus a *state function*, suggesting that s might have a physical significance, because ρ and e have. We thus may decide to introduce a quantity s defined this way, but the basic questions remains: (i) What is the physical significance of s? (ii) Why is $ds \geq 0$? What is the entropy non-perfect gas in which case s may not be a state function?

To further exhibit the connection between the classical and new forms of the 2nd Law, we observe that by the chain rule,

$$\rho D_u s = \frac{\rho}{e}D_u e - \gamma D_u\rho = \frac{1}{T}(D_u e + \gamma\rho T\nabla\cdot u) = \frac{1}{T}(D_u e + e\nabla\cdot u + (\gamma-1)\rho T\nabla\cdot u)$$

since by mass conservation $D_u\rho = -\rho\nabla\cdot u$. It follows that the entropy $S = \rho s$ satisfies

$$\dot{S} + \nabla\cdot(Su) = \rho D_u s = \frac{1}{T}(\dot{e} + \nabla\cdot(eu) + p\nabla\cdot u) = \frac{1}{T}R_e(\hat{u}). \qquad \textbf{(38.13)}$$

A solution $\hat{u}$ of the regularized Euler equations (38.3) thus satisfies

$$\dot{S} + \nabla\cdot(Su) = \frac{\nu}{T}|\nabla u|^2 \geq 0 \quad \text{in } Q, \qquad \textbf{(38.14)}$$

where $S = \rho\log(e\rho^{-\gamma})$. In particular, in the case of the Joule experiment with T the same in the initial and final states, we have $s = \gamma\log(V)$ showing an increase of entropy in the final state with larger volume.

We sum up by noting that the classical and new form of the second law effectively express the same inequality $ds \geq 0$ or $Tds \geq 0$. The new 2nd law is expressed in terms of the fundamental concepts of of kinetic energy, heat energy and work without resort to any form of entropy and statistical mechanics with all its complications. Of course, the new 2nd Law readily extends to the case of a general gas.

38.12 EG2

EG2 in cG(1)cG(1)-form for the Euler equations (38.2), reads: Find $\hat{u} = (\rho, m, \epsilon) \in V_h$ such that for all $(\bar{\rho}, \bar{u}, \bar{\epsilon}) \in W_h$

$$\begin{aligned} ((R_\rho(\hat{u}), \bar{\rho})) + ((hu \cdot \nabla\rho, u \cdot \nabla\bar{\rho})) &= 0, \\ ((R_m(\hat{u}), \bar{u})) + ((hu \cdot \nabla m, u \cdot \nabla\bar{u})) + (\nu_{sc}\nabla u, \nabla\bar{u})) &= 0, \\ ((R_\epsilon(\hat{u}), \bar{e})) + ((hu \cdot \nabla\epsilon, u \cdot \nabla\bar{\epsilon})) &= 0, \end{aligned} \qquad (38.15)$$

where V_h is a trial space of continuous piecewise linear functions on a space-time mesh of size h satisfying the initial condition $\hat{u}(0) = \hat{u}^0$ with $u \in V_h$ defined by nodal interpolation of $\frac{m}{\rho}$, and W_h is a corresponding test space of function which are continuous piecewise linear in space and piecewise constant in time, all functions satisfying the boundary condition $u \cdot n = 0$ at the nodes on Γ. Further, $((\cdot, \cdot))$ denotes relevant $L_2(Q)$ scalar products, and $\nu_{sc} = h^2|R_m(\hat{u})|$ is a residual dependent *shock-capturing viscosity*, see (Nazarov, 2009). We here use the conservation equation for the total energy ϵ rather than for the internal energy e.

EG2 combines a weak satisfaction of the Euler equations with a weighted least squares control of the residual $R(\hat{u}) \equiv (R_\rho(\hat{u}), R_m(\hat{u}), R_e(\hat{u}))$ and thus represents a midway between the Scylla of weak solution and Charybdis of least squares strong solution.

38.13 The 2nd Law for EG2

Subtracting the mass equation with $\bar{\rho}$ a nodal interpolant of $\frac{|u|^2}{2}$ from the momentum equation with $\bar{u} = u$ and using the heat energy equation with $\bar{e} = 1$, we obtain the following 2nd Law for EG2 (up to a $\sqrt{h}$-correction controlled by the shock capturing viscosity (**?**):

$$\dot{K} = W - D_h, \quad \dot{E} = -W + D_h, \qquad (38.16)$$

where

$$D_h = ((h\rho u \cdot \nabla u, u \cdot \nabla u)). \qquad (38.17)$$

For solutions with turbulence/shocks, $D_h > 0$ expressing an irreversible transfer of kinetic energy into heat energy, just as above for regularized solutions. We note that in EG2 only the momentum equation is subject to viscous regularization, since D_h expresses a penalty on $u \cdot \nabla u$ appearing in the momentum residual.

38.14 The Stabilization in EG2

The stabilization in EG2 is expressed by the dissipative term D_h which can be viewed as a weighted least squares control of the term $\rho u \cdot \nabla u$ in the momentum residual. The rationale is that least squares control of a part of a residual which is large, effectively may give control of the entire residual, and thus EG2 gives a least squares control of the momentum residual. But the EG2 stabilization does not correspond to an ad hoc viscosity, as in classical regularization, but to a form of penalty arising because Euler residuals of turbulent/shock solutions are not pointwise small. In particular the dissipative mechanism of EG2 does not correspond to a simple shear viscosity, but rather to a form of "streamline viscosity" preventing fluid particles from colliding while allowing strong shear.

38.15 EG2 Implementation in FEniCS

The FEniCS implementation of EG2 is done in Unicorn (**?**) with source code displayed in Figs. Fig 38.8 and 38.9.

```
cell = "tetrahedron"
scalar = FiniteElement("Lagrange", cell, 1)
vector = VectorElement("Lagrange", cell, 1)
constant_scalar = FiniteElement("Discontinuous␣Lagrange", cell, 0)
constant_vector = VectorElement("Discontinuous␣Lagrange", cell, 0)

TH = MixedElement([scalar,vector,scalar])
d = scalar.cell_dimension()        # Dimension of domain

(v1, v2, v3) = TestFunctions(TH)   # test basis function
(rho, m, e) = TrialFunctions(TH)   # trial basis function
(rho0, m0, e0) = Functions(TH)     # solution from previous time step

P  = Function(scalar)              # pressure
U  = Function(constant_vector)     # velocity to be computed in the
                                   #solver

delta  = Function(constant_scalar) # stabilization parameter
nu_rho = Function(constant_scalar) # shock capturing parameter for rho
nu_m   = Function(constant_vector) # shock capturing parameter for m
nu_e   = Function(constant_scalar) # shock capturing parameter for e

u      = Function(vector)          # velocity to be computed in the
                                   #solver

k  = Constant(cell)                # time step

#------------------------------
# Galerkin discretization of bilinear form for the density
a1_a = v1*rho*dx - k*0.5*dot(grad(v1),U)*rho*dx

# Stabilization of bilinear form for the density
S1_a = k*0.5*delta*dot(grad(v1),U)*dot(U, grad(rho))*dx + \
      k*0.5*nu_rho*dot(grad(v1),grad(rho))*dx

# Galerkin discretization of linear form for the density
a1_L = v1*rho0*dx + k*0.5*dot(grad(v1),U)*rho0*dx

# Stabilization of linear form for the density
S1_L = - k*0.5*delta*dot(grad(v1),U)*dot(U, grad(rho0))*dx - \
      k*0.5*nu_rho*dot(grad(v1),grad(rho0))*dx
```

Figure 38.8: Source code for the choice of the function spaces and functions, bilinear and linear forms for the conservation of mass.

```
#------------------------------
a2_a = 0
S2_a = 0
a2_L = 0
S2_L = 0

for i in range(0, d):
    # Galerkin discretization of bilinear form for the momentum m_i
    a2_a += v2[i]*m[i]*dx - k*0.5*dot(grad(v2[i]),U)*m[i]*dx

    # Stabilization of bilinear form for the momentum
    S2_a += k*0.5*delta*dot(grad(v2[i]),U)*dot(U, grad(m[i]))*dx + \
        k*0.5*nu_m[i]*dot(grad(v2[i]),grad(m[i]))*dx

    # Galerkin discretization of linear form for the momentum
    a2_L += v2[i]*m0[i]*dx + k*0.5*dot(grad(v2[i]),U)*m0[i]*dx

    # Stabilization of linear form for the momentum
    S2_L += -k*0.5*delta*dot(grad(v2[i]),U)*dot(U, grad(m0[i]))*dx - \
        k*0.5*nu_m[i]*dot(grad(v2[i]),grad(m0[i]))*dx

#------------------------------
# Galerkin discretization of bilinear form for the energy
a3_a = v3*e*dx - k*0.5*dot(grad(v3),U)*e*dx

# Stabilization of bilinear form
S3_a = k*0.5*delta*dot(grad(v3),U)*dot(U, grad(e))*dx + \
       k*0.5*nu_e*dot(grad(v3),grad(e))*dx

# Galerkin discretization of linear form for the density
a3_L = v3*e0*dx + k*0.5*dot(grad(v3),U)*e0*dx + k*dot(grad(v3),U)*P*dx

# Stabilization of linear form
S3_L = - k*0.5*delta*dot(grad(v3),U)*dot(U,grad(e0))*dx - \
       k*0.5*nu_e*dot(grad(v3),grad(e0))*dx

#------------------------------
# Weak form to the Euler equations:
a = a1_a + S1_a + a2_a + S2_a + a3_a + S3_a
L = a1_L + S1_L + a2_L + S2_L + a3_L + S3_L
```

Figure 38.9: Source code for bilinear and linear forms for the conservation of momentum and energy.

CHAPTER 39

Automated Testing of Saddle Point Stability Conditions

By Marie E. Rognes

Chapter ref: **[rognes]**

39.1 Introduction

Over the last five decades, there has been a substantial body of research on the theory of mixed finite element methods. Mixed finite element methods are finite element methods where two or more finite element spaces are used to approximate separate variables. These methods have often been applied to saddle point problems arising from constrained minimization problems. Examples include the Stokes equations, the equations of Darcy flow (or the mixed Laplacian) or the Hellinger-Reissner formulation for linear elasticity. For equations involving several variables, and where elimination of any of the variables is not a viable option, the usefulness of such methods is evident. For other equations, discretizations based on the introduction of additional variables may have improved properties.

For any discretization of a variational problem, stability is crucial to ensure well-posedness. For coercive problems, the discrete stability may often be easily ensured. For mixed discretizations of saddle point problems on the other hand, stability may be a nontrivial affair. Indeed, the mixed finite element spaces must usually be carefully chosen. The stability theory for mixed finite element discretizations originates from the work of Babuška (Babuška, 1972/73) and Brezzi (Brezzi, 1974) in the early 1970's. Brezzi established two conditions ensuring the stability of a mixed finite element discretization of a canonical saddle point problem. Since then, many papers (and books) have been devoted to the identification and construction of specific stable mixed finite elements for specific saddle point problems (Arnold et al., 2006c, Brezzi and Falk, 1991, Brezzi and Fortin, 1991, Brezzi et al., 1985c, Raviart and Thomas, 1977, Taylor and Hood, 1973). Some of the analytical results are well known, such as the stability of the Taylor-Hood elements for the Stokes equations (Brezzi and Falk, 1991, Taylor and Hood, 1973). Other results, such as the reduced stability of the $\mathrm{P}_1^c(\mathbb{V})\times\mathrm{P}_0^d$ elements on criss-cross triangulations for the mixed Laplacian (Boffi et al., 2000), may be less so.

The goal of this note is to demonstrate that the process of numerically examining the stability of any given discretization can be automated. For a given discretization, the Brezzi constants are computable through a set of eigenvalue problems. These eigenvalue problems have previously been used to numerically study the stability of certain discretizations (Arnold and Rognes, 2009, Chapelle and Bathe, 1993, Qin, 1994). However, automation of this task has not been previously considered in the literature. A secondary aim is to show that the automation process is fairly easy given a software framework supporting the following components: a suitable range of different finite element spaces, easy support of bilinear forms defining equations and inner products, and finally, a linear algebra backend with support for generalized, possibly singular, eigenvalue problems. The components of the FEniCS project (FEniCS) provide these tools.

An automated stability tester provides several advantages. First, the notion of saddle point stability goes from something rather abstract to something rather hands-on. Moreover, even a novice user can easily get an overview of the available stable (or unstable) finite elements for a given equation. For research purposes, it provides a tool for the careful examination of discretizations that have stability properties depending on the tessellation structure. In particular, this framework has been used to study the stability properties of Lagrange elements for the mixed Laplacian (Arnold and Rognes, 2009).

This paper is organized as follows. For motivational purposes, a simple example illustrating the importance of discrete stability is presented in Section 39.2. The subsequent two sections summarize the discrete stability theory of Babuška and Brezzi and how the stability constants involved can be computed through a set of eigenvalue problems. In Section 39.5, a strategy for the automation of numerical stability testing is presented. In particular, a light-weight python module, ASCoT (Rognes, 2009), constructed on top of PyDOLFIN (Logg and Wells, 2009), is described. This module is freely available as a FEniCS Application at `https://launchpad.net/ascot`. The use and capabilities of this framework are demonstrated when applied to two classical examples: the mixed Laplacian and the Stokes equations in Section 39.6. Finally, Section 39.7 provides some concluding remarks and a discussion of limitations.

39.2 Why does discrete stability matter?

The following simple example illustrates that discrete stability is indeed crucial for the approximation of saddle point problems. Let $\Omega = (0,1)^2$ be the unit square in $\mathbb{R}^2$, and take $f = -2\pi^2 \sin(\pi x)\sin(\pi y)$. Consider the following mixed formulation of the Poisson problem with homogeneous Dirichlet boundary conditions: for the given data $f \in L^2(\Omega)$, find $\sigma \in H(\mathrm{div}, \Omega)$, and $u \in L^2(\Omega)$ such that

$$\begin{aligned} \langle \sigma, \tau \rangle + \langle \operatorname{div} \tau, u \rangle &= 0 \qquad \forall\, \tau \in H(\mathrm{div}, \Omega), \\ \langle \operatorname{div} \sigma, v \rangle &= \langle f, v \rangle \quad \forall\, v \in L^2(\Omega). \end{aligned} \tag{39.1}$$

This problem is well-posed: such solutions exist, are unique and depend continuously on the given data. In particular, $u = \sin(\pi x)\sin(\pi y)$ and $\sigma = \operatorname{grad} p$ solve (39.1).

Next, let $\mathcal{T}_h$ be a uniform triangulation of the unit square that is formed by dividing the domain into $n \times n$ sub-squares (with h the maximal triangle diameter) and dividing each square by the diagonal with positive slope. Given a pair of finite element spaces $\Sigma_h \times V_h$ defined relative to this tessellation, the equations (39.1) can be discretized in the standard manner: find $\sigma_h \in \Sigma_h$ and $u_h \in V_h$ such that

$$\begin{aligned} \langle \sigma_h, \tau \rangle + \langle \operatorname{div} \tau, u_h \rangle &= 0 \qquad \forall\, \tau \in \Sigma_h, \\ \langle \operatorname{div} \sigma_h, v \rangle &= \langle f, v \rangle \quad \forall\, v \in V_h. \end{aligned} \tag{39.2}$$

The final question becomes what finite element spaces Σ_h and V_h to choose. As we shall see, the well-posedness of the discrete problem will heavily rely on the choice of spaces.

First, let us consider a naive choice; namely, taking the space of continuous piecewise linear vector fields defined relative to $\mathcal{T}_h$ for the space Σ_h and the space of continuous piecewise linears for V_h. This choice turns out to be a rather bad one: the finite element matrix associated with this pair will be singular! Hence, there does not exist a discrete solution (σ_h, u_h) with this choice of $\Sigma_h \times V_h$.

As a second attempt, we keep the space of continuous piecewise linear vector fields for Σ_h, but replace the previous space V_h by the space of piecewise constant functions. This pair might appear to be a more attractive alternative: there does indeed exist a discrete solution (σ_h, u_h). However, the discrete solution is not at all satisfactory. In particular, the approximation of the scalar variable u_h is highly oscillatory, cf. Figure 39.1(a), and hence it is a poor approximation to the correct solution.

The above two alternatives give unsatisfactory results because the discretizations defined by the element spaces are both unstable. A stable low order element pairing is the combination of the lowest order Raviart-Thomas elements and the space of piecewise constants (Raviart and Thomas, 1977). The corresponding u_h approximation is plotted in Figure 39.1(b). This approximation looks qualitatively correct.

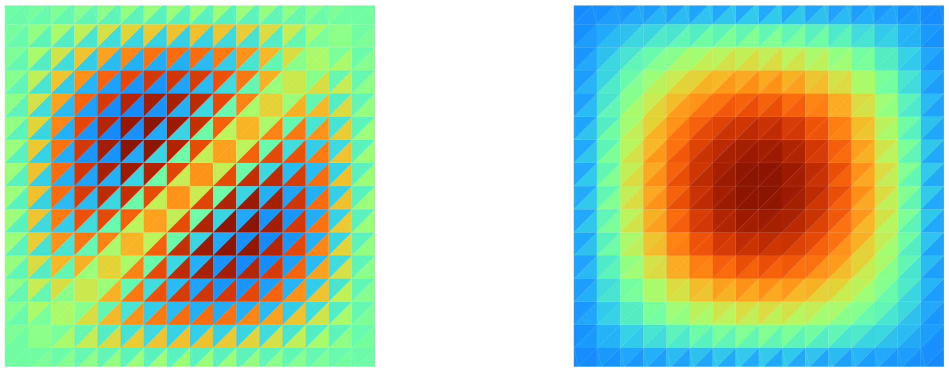


Figure 39.1: The scalar variable approximation u_h for two choices of mixed finite element spaces for the mixed Laplacian. The data are as defined immediately above (39.1). The element spaces are $\mathrm{P}_1^c(\mathbb{V}) \times \mathrm{P}_0^d$ in 39.1(a) and $\mathrm{RT}_1 \times \mathrm{P}_0^d$ in 39.1(b). (The scales are less relevant for the current purpose and have therefore been omitted.)

The reason for the instabilities of the first two choices, and the stability of the third choice, may not be immediately obvious. The goal of this note is to construct a framework that automates this stability identification procedure, by characterizing the stability properties of a finite element discretization automatically and accurately. We will return to this example in Section 39.6 where we give a more careful characterization of the stability properties of the above sample elements.

39.3 Discrete stability

In order to automatically characterize the stability of a discretization, we need a precise definition of discrete stability and preferably conditions for such to hold. In this section, the Babuška and Brezzi stability conditions are described and motivated in the general abstract setting. The material presented here is largely taken from the classical references (Babuška, 1972/73, Brezzi, 1974, Brezzi and Fortin, 1991).

For a Hilbert space W, we denote the norm on W by $\| \cdot \|_W$ and the inner product by $\langle \cdot, \cdot \rangle_W$. Assume that c is a symmetric, bilinear form on W and that l is a continuous, linear form on W. We will consider the following canonical variational problem: find $u \in W$ such that

$$c(v, u) = l(v) \quad \forall\, v \in W. \tag{39.3}$$

Assume that c is continuous; that is, there exists a positive constant C such that

$$|c(v, u)| \le C \, \|v\|_W \|u\|_W \quad \forall\, u, v \in W.$$

If additionally there exists a positive constant γ such that

$$c(u, u) \geq \gamma \|u\|_W^2,$$

the form c is by definition coercive. This is indeed the case for many variational formulations of partial differential equations arising from standard minimization problems. On the other hand, for many constrained minimization problems, such as those giving rise to saddle point problems, the corresponding c is not coercive. Fortunately, the coercivity condition is sufficient, but not necessary. A weaker condition suffices: there exists a positive constant γ such that

$$0 < \gamma = \inf_{0 \neq u \in W} \sup_{0 \neq v \in W} \frac{|c(v, u)|}{\|v\|_W \|u\|_W}. \tag{39.4}$$

If the continuous c satisfies (39.4), there exists a unique $u \in W$ solving (39.3) (Babuška, 1972/73).

Now, we turn to consider discretizations of (39.3). Let $W_h \subset W$ be a finite dimensional subspace, and consider the discrete problem: find $u_h \in W_h$ such that

$$c(v, u_h) = l(v) \quad \forall\, v \in W_h. \tag{39.5}$$

For the discrete system to be well-posed, analogous conditions as for the continuous case must be satisfied. Note that c restricted to W_h is continuous a forteriori. However, the discrete analogue of (39.4) does not trivially hold. In order to guarantee that (39.5) has a unique solution, we must also have that there exists a positive constant γ_h such that

$$0 < \gamma_0 \leq \gamma_h = \inf_{0 \neq u \in W_h} \sup_{0 \neq v \in W_h} \frac{|c(v, u)|}{\|v\|_W \|u\|_W}. \tag{39.6}$$

Moreover, in order to have uniform behaviour in the limit as $h \to 0$, we must have that $\gamma_h \geq \gamma_0 > 0$ for all $h > 0$; that is, that γ_h is bounded from below independently of h (Babuška, 1972/73).

The condition (39.6) has a simple interpretation in the linear algebra perspective. Taking a basis $\{\phi_i\}_{i=1}^n$ for W_h, in combination with the ansatz $u_h = u_j \phi_j$, we obtain the standard matrix formulation of (39.5):

$$C_{ij} u_j = l(\phi_i) \quad i = 1, \ldots, n,$$

where $C_{ij} = c(\phi_i, \phi_j)$. The Einstein notation, in which summation over repeated indices is implicitly implied, has been used here. This system will have a unique solution if the matrix C is non-singular, or equivalently, if the eigenvalues of C are non-zero. In the special case where c is coercive, all eigenvalues will in fact be positive. Moreover, we must ensure that the generalized eigenvalues (generalized with respect to the inner product on W) do not approach zero as $h \to 0$. This is precisely what is implied by the condition (39.6).

39.3.1 Stability conditions for saddle point problems

We now turn to consider the special case of abstract saddle point problems. In this case, the stability condition (39.6) can be rephrased in an alternative, but equivalent form.

Assume that V and Q are Hilbert spaces, that a is a continuous, bilinear form on $V \times V$, that b is a continuous, bilinear form on $V \times Q$, and that l is a continuous linear form on $V \times Q$. A saddle point problem has the following canonical form: find $u \in V$ and $p \in Q$ such that

$$a(v, u) + b(v, p) + b(u, q) = l((v, q)) \quad \forall\, v \in V, q \in Q. \tag{39.7}$$

The system (39.7) is clearly a special case of (39.3) with the following identifications: let $W = V \times Q$, endow the product space with the norm $\|(v, q)\|_W = \|v\|_V + \|q\|_Q$, and label

$$c((v, q), (u, p)) = a(v, u) + b(v, p) + b(u, p).$$

Assuming that the condition (39.4) is satisfied, the above system admits a unique solution $(u, p) \in V \times Q$.

As in the general case, we aim to discrete (39.7), but now using a pair of conforming finite element spaces V_h and Q_h. Letting $W_h = V_h \times Q_h$, we obtain the following special form of (39.5): find $u_h \in V_h$ and $p_h \in Q_h$ satisfying:

$$a(v, u_h) + b(v, p_h) + b(u_h, q) = l((v, q)) \quad \forall\, v \in V_h, q \in Q_h. \tag{39.8}$$

Again, the well-posedness of the discrete problem follows from the general theory. Applying the definition of (39.6) to (39.7), we define the Babuška constant γ_h:

$$\gamma_h = \inf_{0 \neq (u,p) \in W_h} \sup_{0 \neq (v,q) \in W_h} \frac{|a(v, u) + b(v, p) + b(u, q)|}{(\|u\|_V + \|p\|_Q)(\|v\|_V + \|q\|_Q)} \tag{39.9}$$

In particular, the discrete problem is well-posed if the Babuška stability condition holds; namely, if $\gamma_h \geq \gamma_0 > 0$ for any $h > 0$.

The previous deliberations simply summarized the general theory applied to the particular variational form defined by (39.7). However, the special structure of (39.7) also offers an alternative characterization. The single Babuška stability condition can be split into a pair of stability conditions as follows (Brezzi, 1974). Define

$$\alpha_h = \inf_{0 \neq u \in Z_h} \sup_{0 \neq v \in Z_h} \frac{a(u, v)}{\|u\|_V \|v\|_V}, \tag{39.10}$$

$$\beta_h = \inf_{0 \neq q \in Q_h} \sup_{0 \neq v \in V_h} \frac{b(v, q)}{\|v\|_V \|q\|_Q}, \tag{39.11}$$

where

$$Z_h = \{v \in V_h \,|\, b(v, q) = 0 \quad \forall\, q \in Q_h\}. \tag{39.12}$$

We shall refer to α_h as the Brezzi coercivity constant and β_h as the Brezzi inf-sup constant. The Brezzi stability conditions state that these must stay bounded above zero for all $h > 0$. The Brezzi conditions are indeed equivalent to the Babuška condition (Brezzi, 1974). However, for a specific saddle point problem and a given pair of function spaces, it might be easier to verify the two Brezzi conditions than the single Babuška condition. In summary, these conditions enable a concise characterization of the stability of discretizations of saddle point problems.

Definition 39.1 *A family of finite element discretizations $\{V_h \times Q_h\}_h$ is* stable in $V \times Q$ *if the Brezzi coercivity and inf-sup constants $\{\alpha_h\}_h$ and $\{\beta_h\}_h$ (or equivalently the Babuška inf-sup constants $\{\gamma_h\}_h$) are bounded from below by a positive constant independent of h.*

Throughout this note, the term *a family of discretizations* refers to a collection of finite element discretizations parametrized over a family of meshes.

There are families of discretizations that are not stable in the sense defined above, but possess a certain reduced stability. For a pair $V_h \times Q_h$, we can define the space of spurious modes $N_h \subseteq Q_h$:

$$N_h = \{q \in Q_h \,|\, b(v, q) = 0 \quad \forall\, v \in V_h\}.$$

It can be shown that the Brezzi inf-sup constant is positive if and only if there are no nontrivial spurious modes; that is, if $N_h = \{0\}$ (Qin, 1994). On the other hand, if N_h is nontrivial, one may, loosely speaking, think of the space Q_h as a bit too large. In that case, it may be natural to replace Q_h by the reduced space $N_h^\perp$, the orthogonal complement of N_h in Q_h. This idea motivates the definition of the reduced Brezzi inf-sup constant, relating to the stability of $V_h \times N_h^\perp$:

$$\tilde{\beta}_h = \inf_{0 \neq q \in N_h^\perp} \sup_{0 \neq v \in V_h} \frac{b(v, q)}{\|v\|_V \|q\|_Q}, \tag{39.13}$$

and the definition of reduced stable below. By definition, $\tilde{\beta}_h \neq 0$. The identification of reduced stable discretizations can be interesting from a theoretical viewpoint. Further, such could be used for practical purposes after a filtration of the spurious modes.

Definition 39.2 *A family of discretizations $\{V_h \times Q_h\}_h$ is* reduced stable in $V \times Q$ *if the Brezzi coercivity constants $\{\alpha_h\}_h$ and the reduced Brezzi inf-sup constants $\{\tilde{\beta}_h\}_h$ are bounded from below by a positive constant independent of h.*

39.4 Eigenvalue problems associated with saddle point stability

For a given variational problem, the Brezzi conditions provide a method to inspect the stability of a family of conforming discretizations, defined relative to a family of meshes. However, it seems hardly feasible to automatically verify these conditions in their current form. Fortunately and as we shall see in this section, there is an alternative characterization of the Babuška and Brezzi constants: each stability constant will be related to the smallest (in modulus) eigenvalue of an certain eigenvalue problem. The automatic testing of the stability of a given discretization family can therefore be based on the computation and inspection of certain eigenvalues.

We begin by considering the Babuška inf-sup constant for the element pair $V_h \times Q_h$. It can be easily seen that the Babuška inf-sup constant $\gamma_h = |\lambda_{\min}|$ where $\lambda_{\min}$ is the smallest in modulus eigenvalue of the generalized eigenvalue problem (Arnold and Rognes, 2009, Malkus, 1981): find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ such that

$$a(v, u_h) + b(v, p_h) + b(u_h, q) = \lambda\left(\langle v, u_h\rangle_V + \langle q, p_h\rangle_Q\right) \quad \forall\, v \in V, q \in Q. \tag{39.14}$$

By the same arguments, the Brezzi coercivity constant α_h is the smallest in modulus eigenvalue of the following generalized eigenvalue problem: find $0 \neq u_h \in Z_h$ and $\lambda \in \mathbb{R}$ satisfying

$$a(v, u_h) = \lambda\langle v, u_h\rangle_V \tag{39.15}$$

For the spaces V_h and Q_h, a basis is normally known. For Z_h however, this is usually not the case. (If it had been, the space Z_h might have been better to compute with in the first place.) Therefore, the eigenvalue problem (39.15) is not that easily constructed in practice.

Instead, one may consider an alternative generalized eigenvalue problem: find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ satisfying

$$a(v, u_h) + b(v, p_h) + b(u_h, q) = \lambda\langle u_h, v\rangle_V \tag{39.16}$$

It can be shown that the smallest in modulus eigenvalue of the following eigenvalue problem and the smallest in modulus eigenvalue of (39.15) agree (Arnold and Rognes, 2009). Therefore $\alpha_h = |\lambda_{\min}|$ when $\lambda_{\min}$ is the smallest in modulus eigenvalue of (39.16). The eigenvalue problem (39.16) involves the spaces V_h and Q_h and is therefore more tractable. One word of caution however: if there exists a $q \in Q_h$ such that $b(v, q) = 0$ for all $v \in V_h$, then any λ is an eigenvalue of (39.16). Thus, the problem (39.16) is ill-posed if such q exists. The case where such q exists is precisely the case where the Brezzi inf-sup constant is zero.

Finally, the Brezzi inf-sup constant β_h is the square-root of the smallest eigenvalue $\lambda_{\min}$ of the following eigenvalue problem (Malkus, 1981, Qin, 1994): find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ satisfying

$$\langle v, u_h \rangle_V + b(v, p_h) + b(u_h, q) = -\lambda \langle q, p_h \rangle_Q \tag{39.17}$$

The eigenvalues of (39.17) are all non-negative. Any eigenvector associated with a zero eigenvalue corresponds to a spurious mode. Further, the square-root of the smallest non-zero eigenvalue will be the reduced Brezzi inf-sup constant (Qin, 1994).

39.5 Automating the stability testing

The mathematical framework is now in place. For a given variational formulation, given inner product(s), and a family of function spaces, the eigenvalue problem (39.14) or the problems (39.16) and (39.17) can be used to numerically check stability. The eigenvalue problem (39.17) applied to the Stokes equations was used in this context by Qin (Qin, 1994) and Chapelle and Bathe (Chapelle and Bathe, 1993). A fully automated approach has not been previously available though. This is perhaps not so strange, as an automated approach would be rather challenging to implement within many finite element libraries. However, PyDOLFIN provides ample and suitable tools for this task. In particular, the UFL form language, the collection of finite element spaces supported by FIAT/FFC, and the available SLEPc eigensolvers provide the required functionality.

The definition of an abstract saddle point problem (39.7) and the definition of stability of discretizations of such, Definition 39.1, provide a natural starting point. Based on these definitions, the testing of stability relies on the following input.

- The bilinear forms a and b defining a variational saddle point problem.
- The function spaces V and Q through the inner products $\langle \cdot, \cdot \rangle_V$ and $\langle \cdot, \cdot \rangle_Q$.
- A family of finite element function spaces $\{W_h\}_h = \{V_h \times Q_h\}_h$ parametrized over the mesh size h.

We pause to remark that since (39.7) is a special case of the canonical form (39.3), one may consider the Babuška constant only. However, for the analysis of saddle point problems, the separate behaviour of the individual Brezzi constants may be interesting. For this reason, we focus on the Brezzi stability conditions and the decomposed variational form here.

The following strategy presents itself naturally in order to attempt to characterize the stability of a discretization family. With the above information, one can proceed in the following steps

1. For each function space W_h, construct the eigenvalue problems associated with the Brezzi conditions

2. Solve the eigenvalue problems and identify the appropriate eigenvalues corresponding to the Brezzi constants.

3. Based on the behaviour of the Brezzi constants with respect to h, the discretization family should be classified, cf. Definitions 39.1 and 39.2, as

   (a) Stable

   (b) Unstable

   (c) Unstable, but reduced stable

The above strategy is implemented in the automated stability condition tester ASCoT (Rognes, 2009). ASCoT is a python module dependent on PyDOLFIN compiled with SLEPc. It is designed to automatically evaluate the stability of a discretization family, and in particular, the stability of mixed finite element methods for saddle point problems. ASCoT can be imported as any python module:

```
from ascot import *
```

The remainder of this section describes how the afore described strategy is implemented in ASCoT. Emphasis is placed on the form of the input, the construction and solving of the eigenvalue problems, and the classification of stability based on the stability constants.

Before continuing however, it is necessary to point out a limitation of the numerical testing. The mathematical definition of stability is indeed based on taking the limit as $h \to 0$. However, it is hardly feasible to examine an infinite family of function spaces $\{W_h\}_{h\in\mathbb{R}^+}$ numerically. In practice, one can only consider a finite set of spaces $\{W_{h_i}\}_{i\in(0,\dots,N)}$. Therefore, this strategy can only give numerical evidence, which must be interpreted using appropriate heuristics.

39.5.1 Defining input

ASCoT relies on the variational form language defined by UFL and PyDOLFIN for the specification of forms, inner products and function spaces. In order to illustrate, we take the discrete mixed Laplacian introduced in (39.2) as an example.

First and foremost, consider the specification of the forms a and b. Recall that discrete saddle point stability is not a property relating to a single set of function spaces, but rather a property relating to a family of function spaces. In the typical PyDOLFIN approach, forms are specified in terms of basis functions on a single function space. For our purposes, this seems like a less ideal approach.

Instead, to be able to specify the forms independently of the function spaces, we can take advantage of the python λ functionality. For the mixed Laplacian, the forms a and b read $a = a(v, u) = \langle v, u\rangle$ and $b = b(v, q) = \langle \operatorname{div} v, q\rangle$. These should be specified as

```
# Define a and b forms:
a = lambda u, v: dot(u, v)*dx
b = lambda v, q: div(v)*q*dx
```

The above format is advantageous as it separates the definition of the forms from the function spaces. Hence, the user needs not specify basis functions on each of the separate function spaces: ASCoT handles the initialization of the appropriate basis functions.

Second, the inner products $\langle\cdot,\cdot\rangle_V$ and $\langle\cdot,\cdot\rangle_Q$ must be provided. The inner products are bilinear forms and can therefore be viewed as a special case of the above. For the mixed Laplacian, the appropriate inner products are $\langle u, v\rangle_{\mathrm{div}} = \langle v, u\rangle + \langle \operatorname{div} v, \operatorname{div} u\rangle$ and $\langle p, q\rangle_0 = \langle p, q\rangle$. The corresponding code reads

```
# Define inner products:
Hdiv = lambda u, v: (dot(u, v) + div(u)*div(v))*dx
L2 = lambda p, q: dot(p, q)*dx
```

Third, the function spaces have to be specified. In particular, a list of function spaces corresponding to a set of meshes should be defined. For the testing of the mixed function space consisting of continuous piecewise linear vector fields $\mathrm{P}_1^c(\mathbb{V})$, combined with continuous piecewise linears P_1^c, for a set of diagonal triangulations of the unit square, one can do as follows:

```
# Construct a family of mixed function spaces
meshsizes = [2, 4, 6, 8, 10]
meshes = [UnitSquare(n, n) for n in meshsizes]
W_hs = [VectorFunctionSpace(mesh, "CG", 1) + FunctionSpace(mesh, "CG", 1)
        for mesh in meshes]
```

Note that the reliability of the computed stability characterization increases with the number of meshes and their refinement level.

The stability of the above can now be tested. The main entry point function provided by ASCoT is `test_stability`. This function takes three arguments: a (list of) forms, a (list of) inner products and a list of function spaces:

```
result = test_stability((a, b), (Hdiv, L2), W_hs)
```

A `StabilityResult` is returned. The instructions carried out by this function and the properties of the `StabilityResult` are described in the subsequent paragraphs.

39.5.2 Constructing and solving eigenvalue problems

For the testing of saddle point problems, specified by the two forms a and b, it is assumed that the user wants to check the Brezzi conditions. In order to test these conditions, the Brezzi constants; that is, the Brezzi coercivity and Brezzi inf-sup constants, must be computed for each of the function spaces. ASCoT provides functionality for the computation of these constants: the functions `compute_brezzi_coercivity` and `compute_brezzi_infsup`.

Let us take a closer look at the implementation of `compute_brezzi_infsup`: The input consists of the form b, the inner products (m, n), and a function space W_h. The aim is to construct the eigenvalue problem given by (39.17) and then solve this problem efficiently. To accomplish this, the basis functions on the function space W_h are defined first. The left and right-hand sides of the eigenvalue problems are specified through the forms defined by (39.17). These forms are sent to an `EigenProblem`, and the resulting eigenvalues are then used to initialize an `InfSupConstant`. The `InfSupConstant` class is a part of the characterization machinery and will be discussed further in the next subsection.

```python
def compute_brezzi_infsup(b, (m, n), W_h):

    # Define forms for eigenproblem
    (u, p) = TrialFunctions(W_h)
    (v, q) = TestFunctions(W_h)
    lhs = m(v, u) + b(v, p) + b(u, q)
    rhs = - n(q, p)

    # Compute eigenvalues
    eigenvalues = EigenProblem(lhs, rhs).solve()
    return InfSupConstant(W_h.mesh().hmax(), eigenvalues, sqrt)
```

The computation of the Brezzi coercivity constant takes a virtually identical form, only differing in the definition of the left and right hand sides (lhs and rhs). If only a single form c and a single inner product m is specified, the Babuška condition is tested by similar constructs.

The `EigenProblem` class is a simple wrapper class for the DOLFIN `SLEPcEigenSolver`, taking either a single form, corresponding to a standard eigenvalue problem, or two forms, corresponding to a generalized eigenvalue problem. The eigenvalue problems generated by the Babuška and Brezzi conditions are all generalized eigenvalue problems. For both the Brezzi conditions, the right-hand side matrix will always be singular. The left-hand side matrix may or may not be singular depending on the discretization. For the Babuška conditions, the right-hand side matrix should never be singular, however the left-hand side matrix may be.

SLEPc provides a collection of eigenproblem solvers that can handle generalized, possibly singular eigenvalue problems (Hernandez et al., 2005, 2009). The type of eigensolver can be specified through the DOLFIN parameter interface. For our purposes, two solver types are particularly relevant: the 'lapack' and the 'krylov-schur' solvers. The 'lapack' solver is a direct method. This solver is very robust. However, it computes all of the eigenvalues, and it is thus only suited for relatively small problems. In contrast, the krylov-schur method offers the possibility of only computing a given number of eigenvalues. Since the Brezzi constants are related to the eigenvalue closest to zero, it seems meaningful to only compute the eigenvalue of smallest magnitude. This solver is therefore set as the default solver type in ASCoT. Unfortunately, the krylov-schur solver is less robust for singular problems: it may fail to converge. A partial remedy may be to apply a shift-and-invert spectral transform with an appropriate shift factor to the eigenvalue problem. For more details on spectral transformations in SLEPc cf. (Hernandez et al., 2009). ASCoT applies a shift-and-invert transform with a small shift factor by default for the Brezzi and Babuška inf-sup problems.

39.5.3 Characterizing the discretization

After the eigenvalues and thus the stability constants are computed for the family of function spaces, all that remains is to interpret these constants. ASCoT provides three classes intended to represent and interpret the behaviour of the stability constants: `InfSupConstant`, `InfSupCollection` and `StabilityResult`.

An `InfSupConstant` represents a single inf-sup constant. It is initialized using a mesh size h, a set of values, and an optional operator. The values typically correspond to the computed eigenvalues. If supplied, the operator is applied to the eigenvalues. For instance, ASCoT supplies a square-root operator when computing the Brezzi inf-sup constant. The object can return the inf-sup constant and, if computed, the reduced inf-sup constant and the number of zero eigenvalues. The latter two items are most useful for careful analysis purposes.

A collection of `InfSupConstants` forms an `InfSupCollection`. An `InfSupCollection`'s main purpose is to identify whether or not the stability condition associated with the inf-sup constants holds. The method `is_stable` returns a boolean answer. The stability condition will not hold if any of the inf-sup constants is zero, and it will probably not hold if the inf-sup constants seem to decay with the mesh size h. The rate of decay r_i between two subsequent constants c_i and c_{i+1} is defined as:

$$r_i = \frac{\log_2(c_i) - \log_2(c_{i+1})}{\log_2(h_i) - \log_2(h_{i+1})}$$

where h_i is the corresponding mesh size. Currently, ASCoT classifies a discretization as stable if there are no singularities (no zero eigenvalues for all meshes), and the decay rates are below 1 and consistently decrease or the rate corresponding to the finest mesh is less than a given number (0.1 by default).

Finally, the `StabilityResult` class holds a list of possibly several `InfSupCollections`, each corresponding to a separate inf-sup condition, such as the Brezzi coercivity and the Brezzi inf-sup condition. The `StabilityResult` identifies a discretization as stable if all stability conditions are satisfied, and as unstable otherwise.

39.6 Examples

In this section, we apply the automated stability testing framework to two classical saddle point problems: the mixed Laplacian and the Stokes equations. The behaviour of the various mixed finite elements observed in Section 39.2 will be explained and classical analytical results reproduced. The complete code is available from the demo directory of the ASCoT module.

39.6.1 Mixed Laplacian

We can now return to the mixed Laplacian example described in Section 39.2 and inspect the Brezzi stability properties of the element spaces involved, namely $\mathrm{P}_1^c(\mathbb{V}) \times \mathrm{P}_1^c$, $\mathrm{P}_1^c(\mathbb{V}) \times \mathrm{P}_0^d$ and $\mathrm{RT}_1 \times \mathrm{P}_0^d$. The example considered a family of diagonal triangulations of the unit square. The complete code required to test the stability of the first discretization family was presented piecewise in Section 39.5.1. The stability result can be inspected as follows:

```
print result
for condition in result.conditions:
    print condition
```

The following output appears:

```
<Mixed element: (<Mixed element: (<CG1 on a <triangle of degree 1>>,
<CG1 on a <triangle of degree 1>>)>, <CG1 on a <triangle of degree 1>>)>

Not computing Brezzi coercivity constants because of singularity
Discretization family is: Unstable. Singular. Decaying.

InfSupCollection: beta_h
singularities =  [2, 2, 2, 2, 2]
reduced =        [0.56032, 0.35682, 0.24822, 0.18929, 0.15251]
rates  =         [0.651, 0.895, 0.942, 0.968]

Empty InfSupCollection: alpha_h
```

ASCoT characterizes this discretization family as unstable. For the Brezzi inf-sup eigenvalue problems, there are 2 zero eigenvalues for each mesh. Hence, the Brezzi inf-sup constant is zero, and moreover, the element matrix will be singular. This is precisely what we observed in the introductory example: there was no solution to the discrete system of equations. Moreover, the reduced inf-sup constant is also decaying with the mesh size at a rate that seems to be increasing towards $\mathcal{O}(h)$. So, there is no hope of recovering a stable method by filtering out the spurious modes. Since each Brezzi inf-sup constant is zero, the Brezzi coercivity eigenvalue problems are not computationally well-posed, and thus these constants have not been computed.

The second family of elements considered in Section 39.2 was the combination of continuous piecewise linear vector fields and piecewise constants. Using the same code as before, just replacing the finite element spaces, we obtain the following results:

```
<Mixed element: (<Mixed element: (<CG1 on a <triangle of degree 1>>,
<CG1 on a <triangle of degree 1>>)>, <DG0 on a <triangle of degree 1>>)>
Discretization family is: Unstable. Decaying.

InfSupCollection: beta_h
values =          [0.96443, 0.84717, 0.71668, 0.60558, 0.51771]
rates  =          [0.187, 0.413, 0.586, 0.703]

InfSupCollection: alpha_h
values =          [1, 1, 1, 1, 1]
rates  =          [-1.35e-14, 6.13e-14, 3.88e-13, 4.05e-13]
```

Look at the Brezzi inf-sup constants first. In this case, there are no singular values, and hence the Brezzi inf-sup constants are positive. However, the constants seem to decay with the mesh size at increasing rates. Extrapolating, we can suppose that the constants β_h depend on the mesh size h and decay towards zero with h. ASCoT accordingly labels the discretization as unstable. Since there are no singular values, the Brezzi coercivity problem is well-posed. The Brezzi coercivity constants have therefore been computed. We see that the Brezzi coercivity constant is equal to one for all of the meshes tested. This is also easily deduced: the divergence of the velocity space is included in the pressure space and hence the Brezzi coercivity constant is indeed one for all meshes. Since neither constant is singular, we expect the discrete system of equations to be solvable – as we indeed saw in Section 39.2. The problem with this method hence only lies in the decaying Brezzi inf-sup constant. However, the instability did indeed manifest itself in the discrete approximation cf. Figure 39.1(a).

Finally, we can inspect a stable method, namely the lowest order Raviart-Thomas space combined with the space of piecewise constants:

```
<Mixed element: (<RT1 on a <triangle of degree 1>>,
<DG0 on a <triangle of degree 1>>)>
Discretization family is: Stable.

InfSupCollection: beta_h
values =          [0.97682, 0.97597, 0.97577, 0.97569, 0.97566]
rates  =          [0.00126, 0.000508, 0.000265, 0.000162]

InfSupCollection: alpha_h
values =          [1, 1, 1, 1, 1]
rates  =          [5.6e-11, 1.39e-08, 1.64e-08, 2.24e-07]
```

ASCoT characterizes this mixed element method as stable. It is indeed proven so (Raviart and Thomas, 1977). The Brezzi coercivity constant is equal to 1 for all meshes tested and hence bounded from below. The Brezzi inf-sup constant definitely seems to be bounded from below. (The constant will actually converge to the value $\sqrt{2}\pi(1+2\pi^2)^{-1/2}$ cf. (Arnold and Rognes, 2009).) The satisfactory result observed in Figure 39.1(b) is thus agreement with the general theory.

Caveat emptor.

It is worth noting that the stability properties of some mixed elements can vary dramatically. Here is one example: take the combination of continuous linear vector fields and piecewise constants for the mixed Laplacian. As we have seen above, this element family is non-singular on the diagonal mesh family, but the Brezzi inf-sup constants decay. However, if we inspect a family of criss-cross meshes, specified in DOLFIN using

```
meshes = [UnitSquare(n, n, "crossed") for n in meshsizes]
```

with the mesh sizes as before, the results are different:

```
Discretization family is: Unstable. Singular. Reduced stable.

InfSupCollection: beta_h
singularities = [4, 16, 36, 64, 100]
reduced =        [0.97832, 0.97637, 0.97595, 0.97579, 0.97572]
rates  =         [0.00288, 0.00106, 0.000543, 0.000328]
```

For this mesh family, the Brezzi inf-sup constants are zero and thus the method is singular. (In fact, there are n^2 spurious modes for this element on this mesh (Qin, 1994).) However, the reduced Brezzi inf-sup constants seem to be bounded from below, and so the method could theoretically be stabilized by a removal of the spurious modes. For a careful study of the stability of Lagrange elements for the mixed Laplacian on various mesh families cf. (Arnold and Rognes, 2009).

The results may be more different than illustrated above. A truly stable method will be stable for any admissible tessellation family, but there are methods that are stable on some mesh families, but not in general. Therefore, if determining whether a mixed element is appropriate or not, the discretization should be tested on more than a single mesh family.

39.6.2 Stokes

The Stokes equations is another classical and highly relevant saddle point problem. For simplicity, we here consider the following discrete formulation: find the velocity $u_h \in V_h$, and the pressure $p_h \in Q_h$ such that

$$\begin{aligned} \langle \operatorname{grad} u_h, \operatorname{grad} v\rangle + \langle \operatorname{div} v, p_h\rangle = \langle v, f\rangle \qquad \forall\, v \in V_h, \\ \langle \operatorname{div} u_h, q\rangle = 0 \quad \forall\, q \in Q_h. \end{aligned} \tag{39.18}$$

The previous example demonstrated that it is feasible, even easy, to test stability for any given family of discretizations. Taking this a step further, we can generate a set of all available conforming function spaces on a family of meshes, and test the stability of each. With this aim in mind, ASCoT provides some functionality for creating combinations of mixed function spaces given information on the value dimension of the spaces, the polynomial degree, the meshes and the desired regularity. For instance, to generate all available H^1 conforming vector fields of polynomial degree between 1 and 4 matched with L^2 conforming functions of polynomial degrees between 0 and 3 on a given set of meshes, define

```
specifications = {"value_dimension": (2, 1),
                  "degree": (range(1,5), range(4)),
                  "space": ("H1", "L2")}
spaces = create_spaces(meshes, specifications)
```

For the equations (39.18), the Brezzi coercivity condition always holds as long as V_h does not contain the constant functions. Therefore, it suffices to examine the Brezzi inf-sup condition. For simplicity though, we here examine the V_h spaces with no essential boundary conditions prescribed. With spaces generated as above, this can be accomplished as follows:

```
# Define b form
b = lambda v, q: div(v)*q*dx

# Define inner products:
H1 = lambda u, v: (dot(u, v) + inner(grad(u), grad(v)))*dx
L2 = lambda p, q: dot(p, q)*dx

# Test Brezzi inf-sup condition for the generated spaces
for W_hs in spaces:
    beta_hs = [compute_brezzi_infsup(b, (H1, L2), W_h) for W_h in W_hs]
    result = StabilityResult(InfSupCollection(beta_hs, "beta_h"))
```

Finally, ASCoT provides an optimized mode where only the stability of a discretization family is detected and not possible reduced stabilities. This mode is off by default, but can easily be turned on:

```
ascot_parameters["only_stable"] = True
```

Applying the above to the diagonal mesh family used in the previous example and printing those elements that are classified as stable result in the list of mixed elements summarized in Figure 39.2. The first item on this list is the lowest order Taylor-Hood element, while the third and sixth items are the next elements of the Taylor-Hood family: $\mathrm{P}^c_{k+1}(\mathbb{V}) \times \mathrm{P}^c_k$ for $k \geq 1$. These mixed elements are indeed stable for any family of tessellations consisting of more than three triangles (Brezzi and Falk, 1991, Taylor and Hood, 1973). The seventh item on the list is the $\mathrm{P}^c_2(\mathbb{V}) \times \mathrm{P}^d_0$ element (Crouzeix and Raviart, 1973), while the 9'th and 12'th item are the next order elements of the $\mathrm{P}^c_{k+1}(\mathbb{V}) \times \mathrm{P}^d_{k-1}$ family, which again

1. $\mathrm{P}^c_2(\mathbb{V}) \times \mathrm{P}^c_1$
2. $\mathrm{P}^c_3(\mathbb{V}) \times \mathrm{P}^c_1$
3. $\mathrm{P}^c_3(\mathbb{V}) \times \mathrm{P}^c_2$
4. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^c_1$
5. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^c_2$
6. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^c_3$
7. $\mathrm{P}^c_2(\mathbb{V}) \times \mathrm{P}^d_0$
8. $\mathrm{P}^c_3(\mathbb{V}) \times \mathrm{P}^d_0$
9. $\mathrm{P}^c_3(\mathbb{V}) \times \mathrm{P}^d_1$
10. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^d_0$
11. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^d_1$
12. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^d_2$
13. $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^d_3$

Figure 39.2: List of elements identified as satisfying the Brezzi inf-sup condition for the Stokes equations on a family of diagonal triangulations of the unit square.

is truly stable for $k \geq 1$. The 13'th item on this list, $\mathrm{P}^c_4(\mathbb{V}) \times \mathrm{P}^d_3$ is the lowest order Scott-Vogelius element. This element is the lowest order element of the Scott-Vogelius family $\mathrm{P}^c_k(\mathbb{V}) \times \mathrm{P}^d_{k-1}$ for $k \geq 4$. Note that these elements for $k = 1, 2, 3$ are not on the list — as they should not: these lower order mixed elements are indeed unstable on this tessellation family (Qin, 1994). The stability of the remaining elements follow from the previous results: if the Brezzi inf-sup condition holds for a family $\{V_h \times Q_h\}$, by definition it will also hold for the families $\{V_h \times P_h\}$ for $P_h \subseteq Q_h$.

In conclusion, the elements identified are indeed known to be stable, and the list comprises all the stable conforming finite elements for the Stokes equations on this tessellation family that are available in FFC and generated by the `create_spaces` function.

39.7 Conclusion

This note describes an automated strategy for the testing of stability conditions for mixed finite element discretizations. The strategy has been implemented as a very light-weight python module, ASCoT, on top of PyDOLFIN. The implementation is light-weight because of the powerful tools provided by the PyDOLFIN module, in particular the flexible form language provided through UFL/FFC, the availability of arbitrary order mixed finite elements of various families, and the SLEPc eigensolvers.

We have seen that the automated stability tester has successfully identified available stable and unstable elements when applied to the Stokes equations for a diagonal tessellation family. Moreover, the framework has been used to identify previously unknown stability properties for lower order Lagrange elements for the mixed Laplacian (Arnold and Rognes, 2009).

There are however some limitations. First, numerical evidence is not analytical evidence. The tester makes a stability conjecture based on the computed constants. The conjecture may in some cases be erroneous, and the reliability of this conjecture may be low if only a few meshes are considered. Second, solving generalized, singular eigenvalue problems can be nontrivial. For the Brezzi coercivity constants, the krylov-schur solver easily fails to converge even with an applied shift-and-invert spectral transform. In such a case, one must either return to use a lapack-type solver or consider the Babuška constant directly.

References

J. C. Adam, A. Gourdin Serveniere, J.-C. Nédélec, and P.-A. Raviart. Study of an implicit scheme for integrating Maxwell's equations. *Comput. Methods Appl. Mech. Engrg.*, 22(3):327–346, 1980. ISSN 0045-7825.

M. Aechtner. *Arbitrary Lagrangian-Eulerian Finite Element Modelling of the Human Heart*. 2009. Master Thesis, TRITA-CSC-E 2009:022.

M. Ainsworth and J. Oden. A unified approach to a posteriori error estimation using element residual methods. *Numerische Mathematik*, 65(1):23–50, 1993.

M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley and Sons, New York, 2000. ISBN 0-471-29411-X.

B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. Watson. *Molecular Biology of the Cell*. Garland, 4th edition, 2002.

M. S. Alnæs and K.-A. Mardal. *SyFi*, 2009. `http://www.fenics.org/wiki/SyFi/`.

N. Alperin, M. Mazda, T. Lichtor, and S. H. Lee. From cerebrospinal fluid pulsation to noninvasive intracranial compliance and pressure measured by mri flow studies. *Current Medical Imaging Reviews*, 2:117–129, 2006.

T. Arbogast, C. N. Dawson, P. T. Keenan, M. F. Wheeler, and I. Yotov. Enhanced cell-centered finite differences for elliptic equations on general geometry. *SIAM J. Sci. Comput.*, 19(2):404–425 (electronic), 1998. ISSN 1064-8275.

J. Argyris, I. Fried, and D. Scharpf. The TUBA family of plate elements for the matrix displacement method. *The Aeronautical Journal of the Royal Aeronautical Society*, 72:701–709, 1968.

D. Arnold, R. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006a.

D. N. Arnold and F. Brezzi. Mixed and nonconforming finite element methods: implementation, postprocessing and error estimates. *RAIRO Modél. Math. Anal. Numér.*, 19(1):7–32, 1985. ISSN 0764-583X.

D. N. Arnold and R. S. Falk. A uniformly accurate finite element method for the Reissner–Mindlin plate. *SIAM J. Num. Anal.*, 26:1276–1290, 1989. doi: 10.1137/0726074.

D. N. Arnold and M. E. Rognes. Stability of Lagrange elements for the mixed Laplacian. *Calcolo*, 46(4):245–260, 2009.

D. N. Arnold and R. Winther. Mixed finite elements for elasticity. *Numer. Math.*, 92(3):401–419, 2002. ISSN 0029-599X.

D. N. Arnold, F. Brezzi, and J. Douglas, Jr. PEERS: a new mixed finite element for plane elasticity. *Japan J. Appl. Math.*, 1(2):347–367, 1984. ISSN 0910-2043.

D. N. Arnold, R. S. Falk, and R. Winther. Differential complexes and stability of finite element methods. II. The elasticity complex. In *Compatible spatial discretizations*, volume 142 of *IMA Vol. Math. Appl.*, pages 47–67. Springer, New York, 2006b.

D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numer.*, 15:1–155, 2006c. ISSN 0962-4929.

D. N. Arnold, R. S. Falk, and R. Winther. Mixed finite element methods for linear elasticity with weakly imposed symmetry. *Math. Comp.*, 76(260):1699–1723 (electronic), 2007. ISSN 0025-5718.

I. Babuška. The finite element method with Lagrangian multipliers. *Numer. Math.*, 20:179–192, 1972/73. ISSN 0029-599X.

I. Babuška and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Engrg.*, pages 1597–1615, 1978.

B. Bagheri and L. R. Scott. About Analysa. Technical Report TR–2004–09, University of Chicago, Department of Computer Science, 2004.

S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. URL:`http://www.mcs.anl.gov/petsc`, 2001. URL `http://www.mcs.anl.gov/petsc`.

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II — a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4), 2007.

R. Bank and A. Weiser. Some a posteriori error estimators for elliptic partial differential equations. *Mathematics of Computation*, pages 283–301, 1985.

E. Bänsch. An adaptive finite-element strategy for the three-dimensional time-dependent navier-stokes equations. *J. Comput. Appl. Math.*, 36(1):3–28, 1991. ISSN 0377-0427. doi: http://dx.doi.org/10.1016/0377-0427(91)90224-8.

C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *cs/0004015*, Apr. 2000. URL `http://arxiv.org/abs/cs/0004015`. J. Symbolic Computation (2002) 33, 1-12.

E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*. Prentice–Hall, Englewood–Cliffs, 1981.

R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

D. M. Bers. *Excitation-Contraction Coupling and Cardiac Contractile Force*. Kluwert Academic, Dordrecht, The Netherlands, 2nd edition, 2001.

D. J. Beuckelmann and W. G. Wier. Mechanism of release of calcium from sarcoplasmic reticulum of guinea-pig cardiac cells. *J. Physiol.*, 405:233–255, Nov 1988.

J. Bey. Tetrahedral grid refinement. *Computing*, 55:355–378, 1995.

H. B. Bingham, P. A. Madsen, and D. R. Fuhrman. Velocity potential formulations of highly accurate Boussinesq-type models. *Coastal Engineering*, Article in Press, 2008.

D. Boffi, F. Brezzi, and L. Gastaldi. On the problem of spurious eigenvalues in the approximation of linear elliptic problems in mixed form. *Math. Comp.*, 69 (229):121–140, 2000. ISSN 0025-5718.

A. Bossavit. *Computational Electromagnetics: Variational Formulations, Complementarity, Edge Elements*. Academic Press, 1998.

D. Braess. *Finite elements*. Cambridge University Press, Cambridge, third edition, 2007. ISBN 978-0-521-70518-9. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker.

J. Bramble and M. Zlámal. Triangular elements in the finite element method. *Mathematics of Computation*, pages 809–820, 1970.

A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, pages 333–390, 1977.

S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 1994.

S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008. ISBN 978-0-387-75933-3.

F. Brezzi. On the existence, uniqueness and approximation of saddle-point problems arising from lagrangian multipliers. *RAIRO Anal. Numér.*, R–2:129–151, 1974.

F. Brezzi and R. S. Falk. Stability of higher-order Hood-Taylor methods. *SIAM J. Numer. Anal.*, 28(3):581–590, 1991. ISSN 0036-1429.

F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991. ISBN 0-387-97582-9.

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.*, 47(2):217–235, 1985a. ISSN 0029-599X.

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Variable degree mixed methods for second order elliptic problems. *Mat. Apl. Comput.*, 4(1):19–34, 1985b. ISSN 0101-8205.

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.*, 47(2):217–235, 1985c. ISSN 0029-599X.

A. N. Brooks and T. J. R. Hughes. Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 32(1-3):199–259, Sept. 1982. URL `http://www.sciencedirect.com/science/article/B6V29-47X87NJ-FY/1/8827072c`

F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995. ISBN 0201835959.

J. G. Castaños and J. E. Savage. Parallel refinement of unstructured meshes. In *IASTED PDCS*, 1999.

C. A. P. Castigliano. *Théorie de l'équilibre des systèmes élastiques et ses applications*. A.F. Negro ed., Torino, 1879.

O. Certik, F. Seoane, P. Peterson, et al. sympy: Python library for symbolic mathematics, 2009. URL: `http://sympy.org`.

D. Chapelle and K.-J. Bathe. The inf-sup test. *Comput. & Structures*, 47(4-5): 537–545, 1993. ISSN 0045-7949.

G. Chavent and P. Lemmonier. Identification de la non–linearite' d'une equation parabolique quasi–lineare. *Applied Math. and Opt.*, 26:121–162, 1974.

Y. Chen and P. L.-F. Liu. Modified Boussinesq equations and associated parabolic models for water wave propagation. *J. Fluid Mech.*, 288:351–381, 1994.

I. Christie, D. Griffiths, A. Mitchell, and O. Zienkiewicz. Finite element methods for second order differential equations with significant first derivatives. *International Journal for Numerical Methods in Engineering*, 10(6):1389–1396, 1976.

P. Ciarlet. Lectures on the finite element method. *Lectures on Mathematics and Physics, Tata Institute of Fundamental Research, Bombay*, 49, 1975.

P. G. Ciarlet. *Numerical Analysis of the Finite Element Method*. Les Presses de l'Universite de Montreal, 1976.

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam, New York, Oxford, 1978.

P. G. Ciarlet. *The finite element method for elliptic problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002a. ISBN 0-89871-514-8. Reprint of the 1978 original [North-Holland, Amsterdam; MR0520174 (58 #25001)].

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. SIAM, 2002b.

P. G. Ciarlet and P.-A. Raviart. General Lagrange and Hermite interpolation in R^n with applications to finite element methods. *Arch. Rational Mech. Anal.*, 46:177–199, 1972. ISSN 0003-9527.

G. Compère, J.-F. Remacle, J. Jansson, and J. Hoffman. Transient mesh adaptivity applied to large domain deformations. *To appear in Int. J. Numer. Meth. Eng.*, 2009.

COMSOL. Comsol multiphysics, 2009. http://www.comsol.com.

R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.*, pages 1–23, 1943.

M. Crouzeix and R. S. Falk. Nonconforming finite elments for the stokes problem. *Mathematics of Computation*, 52:437–456, 1989. doi: 10.2307/2008475.

M. Crouzeix and P. A. Raviart. Conforming and non-conforming finite element methods for solving the stationary Stokes equations. *R.A.I.R.O Anal. Numer.*, 7:33–76, 1973.

Cython: C-Extensions for Python. Cython: C-extensions for Python. `http://cython.org`.

R. K. Dash and P. Daripa. Analytical and numerical studies of a singularly perturbed Boussinesq equation. *Appl. Math. Comput.*, 126(1):1–30, 2002. ISSN 0096-3003.

D. B. Davidson. *Computational Electromagnetics for RF and Microwave Engineers*. Cambridge University Press, 2005.

T. Davis. Algorithm 832: UMFPACK V4. 3an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

V. Domínguez and F.-J. Sayas. Algorithm 884: A simple Matlab implementation of the Argyris element. *ACM Transactions on Mathematical Software*, 35(2):16:1–16:11, July 2008. URL `http://doi.acm.org/10.1145/1377612.1377620`.

D. Dutykh and F. Dias. Dissipative Boussinesq equations. *Comptes Rendus Mecanique*, 335:559, 2007. URL `doi:10.1016/j.crme.2007.08.003`.

A. P. Engsig-Karup, J. S. Hesthaven, H. B. Bingham, and P. A. Madsen. Nodal DG-FEM solution of high-order Boussinesq-type equations. *J. Engrg. Math.*, 56(3):351–370, 2006. ISSN 0022-0833.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems I: A linear model problem. *SIAM J. Numer. Anal.*, 28, No. 1:43–77, 1991.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems V: Long-time integration. *SIAM J. Numer. Anal.*, 32:1750–1763, 1995a.

K. Eriksson and C. Johnson. Adaptive Finite Element Methods for Parabolic Problems II: Optimal Error Estimates in$ L_\ infty L_2$ and$ L_\ infty L_\ infty$. *SIAM Journal on Numerical Analysis*, 32:706, 1995b.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems IV: Nonlinear problems. *SIAM Journal on Numerical Analysis*, pages 1729–1749, 1995c.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems III: Time steps variable in space. *in preparation*.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numerica*, 4:105–158, 1995.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

K. Eriksson, C. Johnson, and S. Larsson. Adaptive finite element methods for parabolic problems VI: Analytic semigroups. *SIAM J. Numer. Anal.*, 35:1315–1325, 1998.

R. Falgout and U. Yang. Hypre: A library of high performance preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, pages 632–641. Springer-Verlag London, UK, 2002.

C. Felippa. *Refined Finite Element Analysis of Linear and Nonlinear Two-dimensional Structures*. PhD thesis, The University of California at Berkeley, 1966.

FEniCS. FEniCS software collection. `http://www.fenics.org`.

P. Fernandes and M. Raffetto. Characterization of spurious-free finite element methods in electromagnetics. *COMPEL - The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, 21:147–164, 2002. ISSN 0332-1649. URL `http://dx.doi.org/10.1108/03321640210410814`.

C. Franzini-Armstrong, F. Protasi, and V. Ramesh. Shape, size, and distribution of Ca^{2+} release units and couplons in skeletal and cardiac muscles. *Biophys. J.*, 77(3):1528–1539, Sep 1999. URL `http://www.biophysj.org/cgi/content/full/77/3/1528`.

B. G. Galerkin. Series solution of some problems in elastic equilibrium of rods and plates. *Vestnik inzhenerov i tekhnikov*, 19:897–908, 1915.

A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977. ISSN 0022-3654. URL `http://pubs3.acs.org/acs/journals/doilookup?in_doi=10.1021/j100540a008`.

D. C. Grahame. The electrical double layer and the theory of electrocapillarity. *Chem Rev*, 41(3):441–501, Dec 1947.

J. L. Greenstein and R. L. Winslow. An integrative model of the cardiac ventricular myocyte incorporating local control of Ca^{2+} release. *Biophys. J.*, 83(6):2918–2945, Dec 2002. URL `http://www.biophysj.org/cgi/content/full/83/6/2918`.

A. Guia, M. D. Stern, E. G. Lakatta, and I. R. Josephson. Ion concentration-dependence of rat cardiac unitary l-type calcium channel conductance. *Biophys J*, 80(6):2742–2750, Jun 2001. doi: 10.1016/S0006-3495(01)76242-X. URL `http://dx.doi.org/10.1016/S0006-3495(01)76242-X`.

S. Gupta, M. Soellinger, P. Boesiger, D. Pulikako, and V. Kurtcuoglu. Three-dimensional computational modeling of subject -specific cerebrospinal fluid flow in the subarachnoid space. *J. Biomech. Eng.*, 131, 2009.

E. Hairer and G. Wanner. *Solving Ordinary Differential Equations I — Nonstiff Problems*. Springer Series in Computational Mathematics, vol. 8, 1991.

P. Hansbo and M. G. Larson. Discontinuous Galerkin and the Crouzeix–Raviart element: application to elasticity. *Math. Model. Numer. Anal.*, 37:63–72, 2003. doi: 10.1051/m2an:2003020.

J. D. Heiss, N. Patronas, H. L. DeVroom, T. Shawker, R. Ennis, W. Kammerer, A. Eidsath, T. Talbot, J. Morris, E. Eskioglu, and E. H. Oldfield. Elucidating the pathophysiology of syringomyelia. *J. Neurosurg.*, 91:532–562, 1999.

V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, Sept. 2005.

V. Hernandez, J. E. Roman, E. Romero, A. Tomas, and V. Vidal. SLEPc users manual. Technical Report DSIC-II/24/02 - Revision 3.0.0, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2009.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1089014.1089021.

M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.

M. L. Hetland. *Practical Python*. APress, 2002.

B. Hille. *Ion Channels of Excitable Membranes*. Sinauer, Sunderland, MA, 2001.

J. Hoffman. Computation of mean drag for bluff body problems using adaptive dns/les. *SIAM J. Sci. Comput.*, 27(1):184–207, 2005.

J. Hoffman. Adaptive simulation of the turbulent flow past a sphere. *J. Fluid Mech.*, 568:77–88, 2006a.

J. Hoffman. Computation of turbulent flow past bluff bodies using adaptive general galerkin methods: drag crisis and turbulent euler solutions. *Comput. Mech.*, 38:390–402, 2006b.

J. Hoffman. Simulation drag crisis for a sphere using skin friction boundary conditions. In *Proceedings of ECCOMAS CFD*, 2006c.

J. Hoffman. Efficient computation of mean drag for the subcritical flow past a circular cylinder using general galerkin g2. *Int. J. Numer. Meth. Fluids*, 2009.

J. Hoffman and C. Johnson. *Computational Turbulent Incompressible Flow: Applied Mathematics: Body and Soul Vol 4*. Springer-Verlag, 2006a.

J. Hoffman and C. Johnson. A new approach to computational turbulence modeling. *Comput. Methods Appl. Mech. Engrg.*, 195:2865–2880, 2006b.

J. Hoffman and C. Johnson. Blowup of the incompressible euler equations. *BIT Numerical Mathematics*, 48:285–307, 2008a.

J. Hoffman and C. Johnson. Resolution of d'alembert's paradox. *J. Math. Fluid Mech.*, Online First December 2008, 2008b.

J. Hoffman, J. Jansson, and M. Stöckli. Unified continuum modeling of 3d fluid-structure interaction. *SIAM Journal on Scientific Computing (in review)*, 2009.

E. Hofmann, M. Warmuth-Metz, M. Bendszus, and L. Solymosi. Phase-contrast MR imaging of the cervical CSF and spinal cord: volumetric motion analysis in patients with Chiari i malformation. *AJNR Am J Neuroradiol*, 21:151–158, 2000.

Y. Hu and R. Blake. An optimal dynamic load balancing algorithm. Technical Report DL-P95-011, Daresbury Laboratory, Warrington, UK, 1995.

W. Huang and T. Itoh. Complex modes in lossless shielded microstrip lines. *IEEE Transactions on Microwave Theory and Techniques*, 36(1):163–165, 1988.

T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

Hypre. Hypre software package. `http://www.llnl.gov/CASC/hypre/`.

I. M. Wilbers and H. P. Langtangen and Å. Ødegård. Using Cython to Speed up Numerical Python Programs. In B. Skallerud and H. I. Andersson, editors, *Proceedings of MekIT'09*, pages 495–512. NTNU, Tapir Academic Press, 2009. ISBN 978-82-519-2421-4.

M. G. Imhof and A. K. Sharma. Seismostratigraphic inversion: Appraisal, ambiguity, and uncertainty. *Geophysics*, 72(4):R51–R65, 2007.

M. S. Jafri, J. J. Rice, and R. L. Winslow. Cardiac Ca^{2+} dynamics: the roles of ryanodine receptor adaptation and sarcoplasmic reticulum load. *Biophys. J.*, 74(3):1149–68, Mar 1998. URL `http://www.biophysj.org/cgi/content/full/74/3/1149`.

J. Jansson. Performance optimization of unicorn. Technical report, 2009.

N. Jansson. Adaptive Mesh Refinement for Large Scale Parallel Computing with DOLFIN. Master's thesis, Royal Institute of Technology, School of Computer Science and Engineering, 2008. TRITA-CSC-E 2008:051.

N. Jansson and J. Hoffman. A computational study of turbulent flow separation for a circular cylinder using skin friction boundary conditions. In *Proceedings of Workshop for Quality and Reliability of Large-Eddy Simulations II*, 2009.

J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley & Sons, Inc., 2nd edition, 2002.

V. John and P. Knobloch. On spurious oscillations at layers diminishing (sold) methods for convection-diffusion equations: Part i - a review. *Computer Methods in Applied Mechanics and Engineering*, 196(17-20):2197 – 2215, 2007. ISSN 0045-7825. doi: DOI:10.1016/j.cma.2006.11.013. URL `http://www.sciencedirect.com/science/article/B6V29-4MPC439-2/2/e985dbff2`

R. S. Johnson. *A modern introduction to the mathematical theory of water waves*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 1997. ISBN 0-521-59832-X.

R. C. Kirby. FIAT: A new paradigm for computing finite element basis functions. *ACM Trans. Math. Software*, 30:502–516, 2004.

R. C. Kirby. *FIAT*, 2006. URL: `http://www.fenics.org/fiat/`.

R. C. Kirby. FErari. `http://www.fenics.org/ferari/`.

R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006. ISSN 0098-3500.

R. C. Kirby and A. Logg. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Transactions on Mathematical Software*, 35(2):1–18, 2008. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1377612.1377614. submitted 2007-04-02, resubmitted 2007-08-23, accepted 2007-12-07 (communication handled by Rob).

R. C. Kirby and L. R. Scott. Geometric optimization of the evaluation of finite element matrices. *to appear in SIAM J. Sci. Comput.*, 2007.

R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel. Topological optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 28(1):224–240, 2006. ISSN 1064-8275.

X. Koh, B. Srinivasan, H. S. Ching, and A. Levchenko. A 3D Monte Carlo analysis of the role of dyadic space geometry in spark generation. *Biophys. J.*, 90(6):1999–2014, Dec 2006. doi: 10.1529/biophysj.105.065466. URL `http://dx.doi.org/10.1529/biophysj.105.065466`.

J. D. Lambert. *Numerical methods for ordinary differential systems*. John Wiley & Sons Ltd., Chichester, 1991. ISBN 0-471-92990-5.

G. A. Langer and A. Peskoff. Calcium concentration and movement in the diadic cleft space of the cardiac ventricular cell. *Biophys. J.*, 70(3):1169–1182, Mar 1996. URL `http://www.pubmedcentral.gov/articlerender.fcgi?tool=pubmed&pubmedid=878`

M. Langner, D. Cafiso, S. Marcelja, and S. McLaughlin. Electrostatics of phosphoinositide bilayer membranes. theoretical and experimental results. *Biophys. J.*, 57(2):335–349, Feb 1990. URL `http://www.pubmedcentral.gov/articlerender.fcgi?tool=pubmed&pubmedid=215`

H. P. Langtangen. *Python Scripting for Computational Science*. Springer, third edition, 2009a.

H. P. Langtangen. *A Primer on Scientific Programming with Python*. Texts in Computational Science and Engineering, vol 6. Springer, 2009b.

P. Lascaux and P. Lesaint. Some nonconforming finite elements for the plate bending problem. *Rev. Française Automat. Informat. Recherche Operationnelle RAIRO Analyse Numérique*, 9(R-1):9–53, 1975. ISSN 0399-0516.

P. Lax and A. Milgram. Parabolic equations. *Annals of Mathematics Studies*, 33: 167–190, 1954.

J.-F. Lee, D.-K. Sun, and Z. J. Cendes. Full-wave analysis of dielectric waveguides using tangential vector finite elements. *IEEE Trans. Microwave Theory Tech.*, 39(8):1262–1271, August 1991.

X. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

P.-F. Liu and S.-B. Woo. Finite element model for modified Boussinesq equations i: Model development. *Journal of Waterway, Port, Coastal and Ocean Engineering*, 130(1):17–28, 2004.

A. Logg. Efficient representation of computational meshes. In B. Skallerud and H. I. Andersson, editors, *MekIT'07*. Tapir Academic Press, 2007. ISBN 9788259122357. http://www.ntnu.no/mekit07, http://butikk.tapirforlag.no/no/node/1073.

A. Logg and G. N. Wells. Dolfin: Automated finite element computing. *ACM Transactions on Mathematical Software*, 2009. URL `http://www.dspace.cam.ac.uk/handle/1810/214787`. submitted 2009-02-13.

F. Loth, M. A. Yardimci, and N. Alperin. Hydrodynamic modeling of cerebrospinal fluid motion within the spinal cavity. *J. Biomech. Eng.*, 123:71–79, 2001.

M. Lutz. *Learning Python*. O'Reilly, third edition, 2007.

M. Lutz. *Programming Python*. O'Reilly, third edition, 2006.

P. A. Madsen and Y. Agnon. Accuracy and convergence of velocity formulations for water waves in the framework of Boussinesq theory. *J. Fluid Mech.*, 477: 285–319, 2003. ISSN 0022-1120.

P. A. Madsen, H. B. Bingham, and H. A. Schäffer. Boussinesq-type formulations for fully nonlinear and extremely dispersive water waves: derivation and analysis. *R. Soc. Lond. Proc. Ser. A Math. Phys. Eng. Sci.*, 459(2033):1075–1104, 2003. ISSN 1364-5021.

D. S. Malkus. Eigenproblems associated with the discrete LBB condition for incompressible finite elements. *Internat. J. Engrg. Sci.*, 19(10):1299–1310, 1981. ISSN 0020-7225.

A. Martelli. *Python in a Nutshell*. O'Reilly, second edition, 2006.

A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.

E. Martini, G. Pelosi, and S. Selleri. A hybrid finite-element-modal-expansion method with a new type of curvilinear mapping for the analysis of microwave passive devices. *IEEE Transactions on Microwave Theory and Techniques*, 51 (6):1712–1717, 2003.

S. G. McLaughlin, G. Szabo, and G. Eisenman. Divalent ions and the surface potential of charged phospholipid membranes. *J. Gen. Physiol.*, 58(6):667–687, Dec 1971. URL `http://www.jgp.org/cgi/reprint/58/6/667`.

Mercurial software package. Mercurial software package. `http://www.selenic.com/mercurial/wiki`.

T. Milhorat and P. A. Bolognese. Tailored operative technique for chiari type i malformations using intraoperative color doppler ultrasonography. *Neurosurgery*, 53(4):899–906, 2003.

T. Milhorat, M. Chou, E. Trinidad, R. Kula, M. Mandell, C. Wolpert, and M. Speer. Chiari i malformations redefined: Clinical and radiographic findings for 364 sympomatic patients, 1999.

J. E. Moreira, G. Almsi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castaos, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith, and G. G. Stewart. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2/3):367–376, 2005.

L. Morley. The triangular equilibrium element in the solution of plate bending problems. *Aero. Quart.*, 19:149–169, 1968.

M. Nazarov. *An adaptive finite element method for the compressible Euler equations*. 2009. Licentiate Thesis.

J.-C. Nédélec. Mixed finite elements in R^3. *Numer. Math.*, 35(3):315–341, 1980. ISSN 0029-599X.

J.-C. Nédélec. A new family of mixed finite elements in R^3. *Numer. Math.*, 50(1): 57–81, 1986. ISSN 0029-599X.

N.Lopes. PhD. report: Relatório do curso de formação avançada: Alguns modelos de propagação de ondas marítimas. *Departamento de Matemática, Faculdade de Ciências da Universidade de Lisboa*, 2007.

N.Lopes, P.Pereira, and L. Trabucho. Provisory title: Improved elevation-potential Boussinesq systems for surface water waves. *To be submitted*.

Numerical Python software package. Numerical Python software package. `http://sourceforge.net/projects/numpy`.

O. Nwogu. Alternative form of boussinesq equations for nearshore wave propagation. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 119(6): 618–638, 1993.

J. Oden and L. Demkowicz. *Applied functional analysis*. CRC press, 1996.

E. H. Oldfield, K. Muraszko, T. H. Shawker, and N. J. Patronas. Pathophyiology of syringomyelia associated with chiari i malformation of the cerebellar tonsils. *Neurosurg.*, 80:3–15, 1994.

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous galerkin methods. *SIAM Journal on Scientific Computing*, 31(2):849–864, 2008. doi: 10.1137/070710032. URL `http://dx.doi.org/10.1137/070710032`.

L. Oliker. PLUM parallel load balancing for unstructured adaptive meshes. Technical Report RIACS-TR-98-01, RIACS, NASA Ames Research Center, 1998.

OpenMP Application Program Interface. OpenMP Application Program Interface. `http://openmp.org`.

G. Pelosi, R. Coccioli, and S. Selleri. *Quick Finite Elements for Electromagnetic Waves*. Artech House, 1998. ISBN 0890068488.

D. H. Peregrine. Long waves on a beach. *Journal of Fluid Mechanics Digital Archive*, (27):815–827, 1967. doi: 10.1017/S0022112067002605.

A. Peskoff, J. A. Post, and G. A. Langer. Sarcolemmal calcium binding sites in heart: II. mathematical model for diffusion of calcium released from the sarcoplasmic reticulum into the diadic region. *J. Membr. Biol.*, 129(1):59–69, Jul 1992. doi: 10.1007/BF00232055. URL `http://dx.doi.org/10.1007/BF00232055`.

P. Peterson. F2PY software package. `http://cens.ioc.ee/projects/f2py2e`.

PETSc software package. PETSc software package. `http://www.anl.gov/petsc`.

G. Pinna, F. Alessandrini, A. Alfieri, M. Rossi, and A. Bricolo. Cerebrospinal fluid flow dynamics study in Chiari I malformation: implications for syrinx formation. *Neurosurg Focus 8*, 3(3), 2000.

pkg-config software package. pkg-config software package. `http://pkg-config.freedesktop.org`.

D. M. Pozar. *Microwave Engineering*. John Wiley & Sons, Inc., third edition, 2005.

Pyrex – a Language for Writing Python Extension Modules. Pyrex – a Language for Writing Python Extension Modules. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex`.

Python programming language. Python programming language. `http://www.python.org`.

J. Qin. *On the Convergence of Some Simple Finite Elements for Incompressible Flows*. PhD thesis, Penn State, 1994.

P.-A. Raviart and J. M. Thomas. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods (Proc. Conf., Consiglio Naz. delle Ricerche (C.N.R.), Rome, 1975)*, pages 292–315. Lecture Notes in Math., Vol. 606. Springer, Berlin, 1977.

Rayleigh. On the theory of resonance. *Trans. Roy. Soc.*, A161:77–118, 1870.

W. Ritz. Über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik. *J. reine angew. Math.*, 135:1–61, 1908.

M. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984. doi: 10.1137/0721042. URL `http://link.aip.org/link/?SNA/21/604/1`.

M.-C. Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *Journal of Computational and Applied Mathematics*, 36(1): 78–89, 1992.

J. C. Rivenæs. Application of a dual–lithology, depth–dependent diffusion equation in stratigraphic simulation. *Basin Research*, 4:133–146, 1992.

J. C. Rivenæs. *A Computer Simulation Model for Siliciclastic Basin Stratigraphy*. PhD thesis, NTH–Trondheim, 1993.

M. Rognes, R. C. Kirby, and A. Logg. Efficient assembly of $h(\mathrm{div})$ and $h(\mathrm{curl})$ conforming finite elements. *SIAM J. Sci. Comput.*, 2008. submitted by Marie 2008-10-24, resubmitted by Marie 2009-05-22.

M. E. Rognes. Automated Stability Condition Tester (ASCoT). `https://launchpad.net/ascot`, 2009.

A. Roldan, V. Haughton, O. Wieben, T. Osswald, and N. Chesler. Characterization of complex CSF hydrodynamics at the cranio-vertebral junction with computational flow analysis: Healthy and Chiari I malformation. *Submitted to Am. J. Neuroradiol.*, 2008.

S. Rüdiger, J. W. Shuai, W. Huisinga, C. Nagaiah, G. Warnecke, I. Parker, and M. Falcke. Hybrid stochastic and deterministic simulations of calcium blips. *Biophys. J.*, 93(6):1847–1857, Sep 2007. doi: 10.1529/biophysj.106.099879. URL `http://dx.doi.org/10.1529/biophysj.106.099879`.

T. F. Russell and M. F. Wheeler. *The Mathematics of Reservoir Simulation*, volume 1 of *Frontiers Applied Mathematics*, chapter Finite element andfinite difference methods for continuous flow in porous media, pages 35–106. SIAM, 1983.

Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

P. Sagaut. *Large-Eddy Simulation for Incompressible Flows — An Introduction*. Springer-Verlag, 2005.

P. Sagaut, S. Deck, and M. Terracol. *Multiscale and multiresolution approaches in turbulence*. Imperial College Press, 2006.

M. Sala, W. F. Spotz, and M. A. Heroux. Pytrilinos: High-performance distributed-memory solvers for python. *ACM Trans. Math. Softw.*, 34(2):1–33, 2008. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1326548.1326549.

K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.

K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker. A performance study of diffusive vs. remapped load-balancing schemes. In *11th Intl. Conference on Parallel and Distributed Computing Systems*, 1998.

H. J. Schroll. *Dual Problems in Depositional Modeling*. Research Report, Simula Research Laboratory, 2008.

SciPy software package. SciPy software package. `http://www.scipy.org`.

H. Si. TetGen. a quality tetrahedral mesh generator and three-dimensional delaunay triangulator. URL:`http://tetgen.berlios.de`, 2007. URL `http://tetgen.berlios.de`.

B. W. Smith, G. J. Chase2, G. M. Shaw, and R. I. Nokes. Simulating transient ventricular interaction using a minimal cardiovascular system model. *Physiological Measurement*, pages 165–179, 2006.

G. S. Smith. *An Introduction to Classical Electromagnetic Radiation*. Cambridge University Press, 1997.

C. Soeller and M. B. Cannell. Numerical simulation of local calcium movements during l-type calcium channel gating in the cardiac diad. *Biophys. J.*, 73(1):97–111, Jul 1997. URL `http://www.pubmedcentral.gov/articlerender.fcgi?tool=pubmed&pubmedid=919`

C. Soeller and M. B. Cannell. Examination of the transverse tubular system in living cardiac rat myocytes by 2-photon microscopy and digital image-processing techniques. *Circ. Res.*, 84(3):266–75, Feb 1999. URL `http://circres.ahajournals.org/cgi/content/full/84/3/266`.

P. Šolín, K. Segeth, and I. Doležel. *Higher-order finite element methods*. Studies in Advanced Mathematics. Chapman & Hall/CRC, Boca Raton, FL, 2004. ISBN 1-58488-438-X. With 1 CD-ROM (Windows, Macintosh, UNIX and LINUX).

M. D. Stern, L. S. Song, H. Cheng, J. S. Sham, H. T. Yang, K. R. Boheler, and E. Ros. Local control models of cardiac excitation-contraction coupling. a possible role for allosteric interactions between ryanodine receptors. *J. Gen. Physiol.*, 113(3):469–489, Mar 1999. URL `http://www.jgp.org/cgi/content/abstract/113/3/469`.

K. Stewartson. D'alembert's paradox. *SIAM Review*, 23(3):308–343, 1981.

G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, 1973.

SWIG software package. SWIG software package. `http://www.swig.org`.

Swiginac Python interface to GiNaC. Swiginac Python interface to GiNaC. `http://swiginac.berlios.de`.

A. J. Tanskanen, J. L. Greenstein, A. Chen, S. X. Sun, and R. L. Winslow. Protein geometry and placement in the cardiac dyad influence macroscopic properties of calcium-induced calcium release. *Biophys. J.*, 92 (10):3379–3396, May 2007. doi: 10.1529/biophysj.106.089425. URL `http://dx.doi.org/10.1529/biophysj.106.089425`.

C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Internat. J. Comput. & Fluids*, 1(1):73–100, 1973. ISSN 0045-7930.

T. Tezduyar and Y. Park. Discontinuity-capturing finite element formulations for nonlinear convection-diffusion-reaction equations. *Comp. Methods Appl. Mech. Eng.*, 59(3):307–325, 1986.

The Python Tutorial. The python tutorial. `http://docs.python.org/tutorial/`.

H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992. doi: 10.1137/0913035. URL `http://link.aip.org/link/?SCE/13/631/1`.

L. Vardapetyan and L. Demkowicz. hp-vector finite elements method for the full-wave analysis of waveguides with no spurious modes. *Electromagnetics*, 22(5): 419–428, July 2002.

R. Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. In *Proceedings of the fifth international conference on Computational and applied mathematics table of contents*, pages 67–83. Elsevier Science Publishers BV Amsterdam, The Netherlands, The Netherlands, 1994.

R. Verfürth. A review of a posteriori error estimation techniques for elasticity problems. *Computer Methods in Applied Mechanics and Engineering*, 176(1-4):419–440, 1999.

VTK software package. VTK software package. `http://www.kitware.com`.

M. Walkley. *A Numerical Method for Extended Boussinesq Shallow-Water Wave Equations*. PhD thesis, The University of Leeds, School of Computer Studies, 1999.

M. Walkley and M. Berzins. A finite element method for the two-dimensional extended Boussinesq equations. *Internat. J. Numer. Methods Fluids*, 39(10): 865–885, 2002. ISSN 0271-2091.

S.-L. Wang, Z.-R. Wu, Y.-L. Cheng, and M. Liu. Effects of surface tension and uneven bottom on surface solitary waves. *Journal of Physics: Conference Series*, 96, 2008.

Weave: Tools for inlining C/C++ in Python. Weave: Tools for inlining C/C++ in Python. `http://scipy.org/Weave`.

G. Wei and J. T. Kirby. Time-dependent numerical code for extended Boussinesq equations. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 121(5):251–261, 1995. doi: 10.1061/(ASCE)0733-950X(1995)121:5(251). URL `http://link.aip.org/link/?QWW/121/251/1`.

G. Wei, J. Kirby, and A. Sinha. Generation of waves in Boussinesq models using a source function method. *Coastal Engineering*, 36:271–279, 1999.

P. Wesseling. *An introduction to multigrid methods*. Wiley Chichester, 1992.

G. B. Whitham. *Linear and nonlinear waves*. Wiley-Interscience [John Wiley & Sons], New York, 1974. Pure and Applied Mathematics.

M. M. Wolf and M. T. Heath. Combinatorial optimization of matrix-vector multiplicaion in finite element assembly. *SIAM J. Sci. Comput.*, 31:2960, 2009.

M. M. Zdravkovich. *Flow Around Circular Cylinders*. Oxford University Press, 2003.

L. Zhang. A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection. Technical Report Preprint ICM-05-09, Institute of Computational Mathematics and Scientific/Engineering Computing, 2005.

M. Zhao, B. Teng, and L. Cheng. A new form of generalized Boussinesq equations for varying water depth. *Ocean Engineering*, 31:2047–2072, 11 2004. URL `http://www.sciencedirect.com/science/article/B6V4F-4D8VFVK-1/2/f20de8fc8`

O. Zienkiewicz and J. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International Journal for Numerical Methods in Engineering*, 24(2), 1987.

O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method — Its Basis and Fundamentals, 6th edition*. Elsevier, 2005, first published in 1967.

APPENDIX A

Notation

The following notation is used throughout this book.

A	–	the *global tensor* with entries $\{A_i\}_{i\in\mathcal{I}}$
A^K	–	the *element tensor* with entries $\{A_i^K\}_{i\in\mathcal{I}_K}$
A^0	–	the *reference tensor* with entries $\{A_{i\alpha}^0\}_{i\in\mathcal{I}_K,\alpha\in\mathcal{A}}$
a	–	a multilinear form
a_K	–	the local contribution to a multilinear form a from a cell K
$\mathcal{A}$	–	the set of *secondary indices*
$\mathcal{B}$	–	the set of *auxiliary indices*
e	–	the *error*, $e = u_h - u$
F_K	–	the mapping from the reference cell K_0 to K
G_K	–	the *geometry tensor* with entries $\{G_K^\alpha\}_{\alpha\in\mathcal{A}}$
$\mathcal{I}$	–	the set $\prod_{j=1}^{\rho}[1, N^j]$ of indices for the global tensor A
$\mathcal{I}_K$	–	the set $\prod_{j=1}^{\rho}[1, n_K^j]$ of indices for the element tensor A^K (*primary indices*)
ι_K	–	the *local-to-global mapping* from $[1, n_K]$ to $[1, N]$
K	–	a *cell* in the mesh $\mathcal{T}$
K_0	–	the *reference cell*
L	–	a linear form (functional) on $\hat{V}$ or $\hat{V}_h$
$\mathcal{L}$	–	the degrees of freedom (linear functionals) on V_h
$\mathcal{L}_K$	–	the degrees of freedom (linear functionals) on $\mathcal{P}_K$
$\mathcal{L}_0$	–	the degrees of freedom (linear functionals) on $\mathcal{P}_0$
N	–	the dimension of $\hat{V}_h$ and V_h
n_K	–	the dimension of $\mathcal{P}_K$
ℓ_i	–	a degree of freedom (linear functional) on V_h
ℓ_i^K	–	a degree of freedom (linear functional) on $\mathcal{P}_K$
ℓ_i^0	–	a degree of freedom (linear functional) on $\mathcal{P}_0$
$\mathcal{P}_K$	–	the local function space on a cell K
$\mathcal{P}_0$	–	the local function space on the reference cell K_0

$P_q(K)$	–	the space of polynomials of degree $\leq q$ on K
r	–	the (weak) residual, $r(v) = a(v, u_h) - L(v)$ or $r(v) = F(u_h; v)$
u_h	–	the finite element solution, $u_h \in V_h$
U	–	the vector of degrees of freedom for $u_h = \sum_{i=1}^{N} U_i \phi_i$
u	–	the exact solution of a variational problem, $u \in V$
$\hat{V}$	–	the test space
V	–	the trial space
$\hat{V}^*$	–	the dual test space, $\hat{V}^* = V_0$
V^*	–	the dual trial space, $V^* = \hat{V}$
$\hat{V}_h$	–	the discrete test space
V_h	–	the discrete trial space
ϕ_i	–	a basis function in V_h
$\hat{\phi}_i$	–	a basis function in $\hat{V}_h$
ϕ_i^K	–	a basis function in $\mathcal{P}_K$
Φ_i	–	a basis function in $\mathcal{P}_0$
z	–	the *dual solution*, $z \in V^*$
$\mathcal{T}$	–	the *mesh*, $\mathcal{T} = \{K\}$
Ω	–	a bounded domain in $\mathbb{R}^d$

List of Authors

► Editor note: *Include author affiliations here.*

► Editor note: *Sort alphabetically by last name, not first.*

- Anders Logg
- Andy R. Terrel
- Bjørn Fredrik Nielsen
- Claes Johnson
- David B. Davidson
- Emil Alf Løvgren
- Evan Lezar
- Garth N. Wells
- Hans Joachim Schroll
- Hans Petter Langtangen
- Ilmar M. Wilbers
- Joakim Sundnes
- Johan Hake
- Johan Hoffman
- Johan Jansson

- Kent-Andre Mardal
- Kristian B. Ølgaard
- Kristian Valen-Sendstad
- Kristoffer Selim
- L. Ridgway Scott
- L. Trabucho
- Luca Antiga
- Marie E. Rognes
- Martin S. Alnæs
- Matthew G. Knepley
- Mehdi Nikbakht
- Murtazo Nazarov
- Niclas Jansson
- N. Lopes
- Oddrun Christine Myklebust
- Ola Skavhaug
- P. Pereira
- Robert C. Kirby
- Susanne Hentschel
- Svein Linge
- Xuming Shan

Index