FFC User Manual

June 20, 2007

Anders Logg

www.fenics.org

Visit `http://www.fenics.org/` for the latest version of this manual.
Send comments and suggestions to `ffc-dev@fenics.org`.

Contents

About this manual

Intended audience

This manual is written both for the beginning and the advanced user. There is also some useful information for developers. More advanced topics are treated at the end of the manual or in the appendix.

Typographic conventions

- Code is written in monospace (typewriter) `like this`.

- Commands that should be entered in a Unix shell are displayed as follows:

```
# ./configure
# make
```

  Commands are written in the dialect of the `bash` shell. For other shells, such as `tcsh`, appropriate translations may be needed.

Enumeration and list indices

Throughout this manual, elements x_i of sets $\{x_i\}$ of size n are enumerated from $i = 0$ to $i = n - 1$. Derivatives in $\mathbb{R}^n$ are enumerated similarly: $\frac{\partial}{\partial x_0}, \frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_{n-1}}$.

Contact

Comments, corrections and contributions to this manual are most welcome and should be sent to

ffc-dev@fenics.org

Chapter 1

Introduction

This chapter has not yet been written. In the meantime, refer to [5, 6] where the algorithms that **FFC** is based on are described in detail.

Chapter 2

Quickstart

This chapter demonstrates how to get started with **FFC**, including downloading and installing the latest version of **FFC**, and compiling Poisson's equation. These topics are discussed in more detail elsewhere in this manual. In particular, see Appendix C for detailed installation instructions and Chapter 5 for a detailed discussion of the form language.

2.1 Downloading and installing FFC

The latest version of **FFC** can be found on the **FEniCS** web page:

```
http://www.fenics.org/
```

The following commands illustrate the installation process, assuming that you have downloaded release `x.y.z` of **FFC**:

```
# tar zxfv ffc-x.y.z.tar.gz
# cd ffc-x.y.z
# sudo python setup.py install
```

Make sure that you download the latest release. You may also need to install the Python packages **FIAT** and NumPy. (See Appendix C for detailed instructions.)

2.2 Compiling Poisson's equation with FFC

The discrete variational (finite element) formulation of Poisson's equation, $-\Delta u = f$, reads: Find $u_h \in V_h$ such that

$$a(v, u_h) = L(v) \quad \forall v \in \hat{V}_h, \tag{2.1}$$

with $(\hat{V}_h, V_h)$ a pair of suitable function spaces (the test and trial spaces). The bilinear form $a : \hat{V}_h \times V_h \to \mathbb{R}$ is given by

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

and the linear form $L : \hat{V}_h \to \mathbb{R}$ is given by

$$L(v) = \int_\Omega v \, f \, \mathrm{d}x. \tag{2.3}$$

To compile the pair of forms (a, L) into code that can called to assemble the linear system $Ax = b$ corresponding to the variational problem (2.1) for a pair of discrete function spaces, specify the forms in a text file with extension `.form`, e.g. `Poisson.form`, as follows:

```
element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

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

The example is given for piecewise linear finite elements in two dimensions, but other choices are available, including arbitrary order Lagrange elements in two and three dimensions.

To compile the pair of forms implemented in the file `Poisson.form`, call the compiler on the command-line as follows:

```
# ffc Poisson.form
```

This will generate the file `Poisson.h` containing low level C++ code in the UFC (Unified Form-assembly Code) format [2, 3]. The generated code can be used by any UFC-based assembler such as **DOLFIN** [4] to assemble the discrete representations (the matrix A and vector b) of the bilinear form a and linear form L of Poisson's equation.

Note that by adding the flag `-l dolfin`, additional **DOLFIN**-specific wrappers are added to the generated code which simplifies the use of the generated code with **DOLFIN**. In particular, the handling of forms depending on coefficients like f in Poisson's equation is simplified.

For further help on the `ffc` command and available command-line options, refer to the **FFC** man page:

```
# man ffc
```

Chapter 3

Command-line interface

The command-line interface of **FFC** is documented by the **FFC** man page:

```
# man ffc
```

A copy of this documentation is included below for convenience.

```
NAME
       FFC - the FEniCS Form Compiler


SYNOPSIS
       ffc  [-h]  [-v]  [-d debuglevel] [-s] [-l language] [-r representation]
       [-f option] [-O] ... input.form ...


DESCRIPTION
       Compile multilinear forms into efficient low-level code.

       The FEniCS Form Compiler FFC accepts as input one or more  files,  each
       specifying  one or more multilinear forms, and compiles the given forms
       into efficent low-level code for automatic assembly of the tensors rep-
       resenting  the multilinear forms. In particular, FFC compiles a pair of
       bilinear and linear forms defining a variational problem into code that
       can be used to efficiently assemble the corresponding linear system.

       By  default, FFC generates code according to the UFC specification ver-
       sion 1.0 (Unified Form-assembly Code, see  http://www.fenics.org/)  but
       this  can  be  controlled  by  specifying  a  different output language
```

```
       (option -l). It is also possible to add new output languages to FFC.

       For a full description of FFC, including a specification  of  the  form
       language  used to define the multilinear forms, see the FFC user manual
       available on the FEniCS web page: http://www.fenics.org/


OPTIONS
       -h, --help
              Display help text and exit.

       -v, --version
              Display version number and exit.

       -d debuglevel, --debug debuglevel
              Specify debug level (default is 0).

       -s, --silent
              Silent mode, no output is printed (same as --debuglevel -1).

       -l language, --language language
              Specify output language, one of 'ufc' (default) or 'dolfin' (UFC
              with a small layer of DOLFIN-specific bindings).

       -r representation, --representation representation
              Specify  representation  for precomputation and code generation,
              one of 'tensor' (default) or 'quadrature' (experimental).

       -f option
              Specify code generation options. The list of  options  available
              depends  on  the  specified  language  (format). Current options
              include -fprecision=n, -fblas and -fno-foo, described in  detail
              below.

       -f precision=n
              Set the number of significant digits to n in the generated code.
              The default value of n is 15.

       -f blas
              Generate code that uses BLAS to compute tensor  products.   This
              option  is currently ignored, but can be used to reduce the code
              size when the BLAS option is  (re-)implemented  in  future  ver-
              sions.

       -f no-foo
              Don't  generate  code  for UFC function with name 'foo'. Typical
              options    include    -fno-evaluate_basis    and    -fno-evalu-
              ate_basis_derivatives  to  reduce the size of the generated code
              when these functions are not needed.

       -O, --optimize
              Generate optimized code using FErari optimizations. This  option
              is  currently  ignored,  but can be used to reduce the operation
              count for assembly  (run-time  for  the  generated  code).  This
              option  requires FErari and should be used with caution since it
              may be very costly  (at  compile-time)  for  other  than  simple
              forms.
```

```
       BUGS

       Send comments, questions, bug reports etc. to ffc-dev@fenics.org.


AUTHOR
       Anders Logg (logg@simula.no)


                                                                                FFC(1)
```

Chapter 4

Python interface

FFC provides a Python interface in the form of a standard Python module. The following example demonstrates how to define and compile the variational problem for Poisson's equation in a Python script:

```
from ffc import *

element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

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

compile([a, L], "Poisson")
```

At the basic level, the only difference between the command-line interface and the Python interface is that one must add the import statement of the **FFC** module and that the function `compile` must be called when using the Python interface.

4.1 The `compile` function

The `compile` function expects a form (see Section 5) or a list of forms as its first argument. It also accepts up to four additional optional arguments:

```
compile(forms, prefix, representation, language, options)
```

The `prefix` argument can be used to control the prefix of the file containing the generated code, which we in the above example set to `"Poisson"`. The suffix `".h"` will be added automatically.

The `representation` argument can be used to control the form representation used for precomputation and code generation. The default value is `"tensor"`, which indicates that the code should be generated based on a tensor representation of the multilinear form as described in [5, 6]. Alternatively, `"quadrature"` may be used to specify that code should be generated based on direct quadrature at run-time (experimental).

The `language` option can be used to control the output language for the generated code. The default value is `"ufc"`, which indicates that code should be generated in the UFC format [2, 3]. Alternatively, `"dolfin"` may be used to generate code according to the UFC format with a small set of additional **DOLFIN**-specific wrappers.

Finally, the `compile` function accepts a dictionary of special code generation options. The default values for these options may be accessed through the variable `FFC_OPTIONS` available in **FFC**.

4.2 Compiling finite elements

The `compile` function may also be used to compile finite elements directly (without associated forms). The following example demonstrates how to generate code for a fifth degree Lagrange finite element on tetrahedra:

```
from ffc import *

element = FiniteElement("Lagrange", "tetrahedron", 5)
compile(element, "P5")
```

Chapter 5

Form language

FFC uses a flexible and extensible language to define and process multilinear forms. In this chapter, we discuss the details of this form language. In the next section, we present a number of examples to illustrate the use of the form language in applications.

5.1 Overview

FFC compiles a given multilinear form

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \to \mathbb{R} \tag{5.1}$$

into code that can be used to compute the corresponding tensor

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r). \tag{5.2}$$

In the form language, a multilinear form is defined by first specifying the set of function spaces, $V_h^1, V_h^2, \ldots, V_h^r$, and then expressing the multilinear form in terms of the basis functions of these function spaces.

A function space is defined in the form language through a `FiniteElement`, and a corresponding basis function is represented as a `BasisFunction`. The

following code defines a pair of basis functions `v` and `u` for a first-order Lagrange finite element on triangles:

```
element = FiniteElement("Lagrange", "triangle", 1)
v = BasisFunction(element)
u = BasisFunction(element)
```

The two basis functions can now be used to define a bilinear form:

```
a = v*D(u, 0)*dx
```

corresponding to the mathematical notation

$$a(v, u) = \int_\Omega v \frac{\partial u}{\partial x_0} \, \mathrm{d}x. \tag{5.3}$$

Note that the order of the argument list of the multilinear form is determined by the order in which basis functions are declared, not by the order in which they appear in the form. Thus, both `a = v*D(u, 0)*dx` and `a = D(u, 0)*v*dx` define the same multilinear form.

The arity (number of arguments) of a multilinear form is determined by the number of basis functions appearing in the definition of the form. Thus, `a = v*u*dx` defines a *bilinear form*, namely $a(v, u) = \int_\Omega v\, u \, \mathrm{d}x$, whereas `L = v*dx` defines a *linear form*, namely $L(v) = \int_\Omega v \, \mathrm{d}x$.

In the case of a bilinear form, the first of the two basis functions is referred to as the *test function* and the second is referred to as the *trial function*. One may optionally use the keywords `TestFunction` and `TrialFunction` to specify the test and trial functions. This has the advantage that the order of specification of the two functions does not matter; the test function will always be the first argument of a bilinear form and correspond to a row in the corresponding assembled matrix. Thus, the example above may optionally be specified as follows:

```
element = FiniteElement("Lagrange", "triangle", 1)
v = TestFunction(element)
u = TrialFunction(element)
```

Not every expression is a valid multilinear form. The following list explains some of the basic rules that must be obeyed in the definition of a form:

- A form must be linear in each of its arguments; otherwise it is not a multilinear form. Thus, `a = v*v*u*dx` is not a valid form, since it is quadratic in `v`.

- The value of a form must be a scalar. Thus, if `v` is a vector-valued basis function (see below), then `L = v*dx` is not a valid form, since the value of the form is not a scalar.

- The integrand of a form must be integrated exactly once. Thus, neither `a = v*u` nor `a = v*u*dx*dx` are valid forms.

5.2 The form language as a Python extension

The **FFC** form language is built on top of Python. This is true both when calling **FFC** as a compiler from the command-line or when calling the **FFC** compiler from within a Python program. Through the addition of a collection of basic data types and operators, **FFC** allows a form to be specified in a language that is close to the mathematical notation. Since the form language is built on top of Python, any Python code is valid in the definition of a form (but not all Python code defines a multilinear form). In particular, comments (lines starting with `#`) and functions (keyword `def`, see Section 5.11 below) are allowed in the definition of a form.

5.3 Basic data types

5.3.1 `FiniteElement`

The data type `FiniteElement` represents a finite element on a triangle or tetrahedron. A `FiniteElement` is declared by specifying the finite element family, the underlying shape and the polynomial degree:

```
element = FiniteElement(family, shape, degree)
```

The argument `family` is a string and possible values include:

- `"Lagrange"` or `"CG"`, representing standard scalar Lagrange finite elements (continuous piecewise polynomial functions);
- `"Discontinuous Lagrange"` or `"CG"`, representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions);
- `"Crouzeix-Raviart"` or `"CR"`, representing scalar Crouzeix–Raviart elements;
- `"Brezzi-Douglas-Marini"` or `"BDM"`, representing vector-valued Brezzi–Douglas–Marini $H(\mathrm{div})$ elements;
- `"Raviart-Thomas"` or `"RT"`, representing vector-valued Raviart–Thomas $H(\mathrm{div})$ elements.

The argument `shape` is a string and possible values include:

- `"triangle"`, representing a triangle in $\mathbb{R}^2$;
- `"tetrahedron"`, representing a tetrahedron in $\mathbb{R}^3$.

The argument `degree` is an integer specifying the polynomial degree of the finite element. Note that the minimal degree for Lagrange finite elements is one, whereas the minimal degree for discontinuous Lagrange finite elements is zero.

Note that more than one `FiniteElement` can be declared and used in the definition of a form. The following example declares two elements, one linear and one quadratic Lagrange finite element:

```
P1 = FiniteElement("Lagrange", "tetrahedron", 1)
P2 = FiniteElement("Lagrange", "tetrahedron", 2)
```

5.3.2 `VectorElement`

The data type `VectorElement` represents a vector-valued element. Vector-valued elements may be created by repeating any finite element (scalar, vector-valued or mixed) a given number of times. The following code demonstrates how to create a vector-valued cubic Lagrange element on a triangle:

```
element = VectorElement("Lagrange", "triangle", 3)
```

This will create a vector-valued Lagrange element with two components. If the number of components is not specified, it will automatically be chosen to be the equal to the cell dimension. Optionally, one may also specify the number of vector components directly:

```
element = VectorElement("Lagrange", "triangle", 3, 5)
```

Note that vector-valued elements may be created from any given element type. Thus, one may create a (nested) vector-valued element with four components where each pair of components is a first degree BDM element as follows:

```
element = VectorElement("BDM", "triangle", 1, 2)
```

5.3.3 MixedElement

The data type `MixedElement` represents a mixed finite element on a triangle or tetrahedron. The function space of a mixed finite element is defined as the direct sum of the function spaces of a given list of elements. A `MixedElement` is declared by specifying a `list` of `FiniteElement`s:

```
mixed_element = FiniteElement([e0, e1, ...])
```

Alternatively, a `MixedElement` can be created as the sum of a pair[1] of `FiniteElement`s. The following example illustrates how to create a Taylor–Hood element (quadratic velocity and linear pressure):

```
P2 = VectorElement("Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = P2 + P1
```

Elements may be mixed at arbitrary depth, so mixed elements can be used as building blocks for creating new mixed elements. In fact, a `VectorElement` just provides a simple means to create mixed elements. Thus, a Taylor–Hood element may also be created as follows:

```
P2 = FiniteElement("Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = (P2 + P2) + P1
```

[1] Note that summing more than two elements will create a nested mixed element. For example `e = e0 + e1 + e2` will correspond to `e = MixedElement([MixedElement([e0, e1]), e2])`.

5.3.4 BasisFunction

The data type `BasisFunction` represents a basis function on a given finite element. A `BasisFunction` must be created for a previously declared finite element (simple or mixed):

```
v = BasisFunction(element)
```

Note that more than one `BasisFunction` can be declared for the same `FiniteElement`. Basis functions are associated with the arguments of a multilinear form in the order of declaration.

For a `MixedElement`, the function `BasisFunctions` can be used to construct tuples of `BasisFunction`s, as illustrated here for a mixed Taylor–Hood element:

```
(v, q) = BasisFunctions(TH)
(u, p) = BasisFunctions(TH)
```

5.3.5 TestFunction and TrialFunction

The data types `TestFunction` and `TrialFunction` are special instances of `BasisFunction` with the property that a `TestFunction` will always be the first argument in a form and `TrialFunction` will always be the second argument in a form (order of declaration does not matter).

For a `MixedElement`, the functions `TestFunctions` and `TrialFunctions` can be used to construct tuples of `TestFunction`s and `TrialFunction`s, as illustrated here for a mixed Taylor–Hood element:

```
(v, q) = TestFunctions(TH)
(u, p) = TrialFunctions(TH)
```

5.3.6 `Function`

The data type `Function` represents a function belonging to a given finite element space, that is, a linear combination of basis functions of the finite element space. A `Function` must be declared for a previously declared `FiniteElement`:

```
f = Function(element)
```

Note that more than one function can be declared for the same `FiniteElement`. The following example declares two `BasisFunction`s and two `Function`s for the same `FiniteElement`:

```
v = BasisFunction(element)
u = BasisFunction(element)
f = Function(element)
g = Function(element)
```

`Function` is used to represent user-defined functions, including right-hand sides, variable coefficients and stabilization terms. **FFC** treats each `Function` as a linear combination of basis functions with unknown coefficients. It is the responsibility of the user or the system for which the form is compiled to supply the values of the coefficients at run-time. In the case of **DOLFIN**, the coefficients are automatically computed from a given user-defined function during the assembly of a form. In the notation of the UFC interface [2, 3], `Function`s are referred to as *coefficients*.

Note that the order in which `Function`s are declared is important. The code generated by **FFC** accepts as arguments a list of functions that should correspond to the `Function`s appearing in the form in the order they have been declared.

For a `MixedElement`, the function `Functions` can be used to construct tuples of `Function`s, as illustrated here for a mixed Taylor–Hood element:

```
(f, g) = Functions(TH)
```

5.3.7 Constant

The data type `Constant` represents a constant scalar value that is unknown at compile-time. A `Constant` is declared for a given cell shape (`"triangle"` or `"tetrahedron"`):

```
c = Constant(shape)
```

`Constant`s are automatically replaced by (discontinuous) piecewise constant `Function`s. The following two declarations are thus equivalent:

```
DG0 = FiniteElement("Discontinuous Lagrange", "triangle", 0)

c0 = Constant("triangle")
c1 = Function(DG0)
```

5.3.8 VectorConstant

The data type `VectorConstant` represents a constant vector value that is unknown at compile-time. A `VectorConstant` is declared for a given cell shape (`"triangle"` or `"tetrahedron"`):

```
c = VectorConstant(shape)
```

`VectorConstant`s are automatically replaced by (discontinuous) vector-valued piecewise constant `Function`s. The following two declarations are thus equivalent:

```
DG0 = VectorElement("Discontinuous Lagrange", "triangle", 0)

c0 = VectorConstant("triangle")
c1 = Function(DG0)
```

5.3.9 Index

The data type `Index` represents an index used for subscripting derivatives or taking components of vector-valued functions. If an `Index` is declared without any arguments,

```
i = Index()
```

a *free* `Index` is created, representing an *index range* determined by the context; if used to subscript a vector-valued `BasisFunction` or a `Function`, the range is given by the number of vector dimensions n, and if used to subscript a derivative, the range is given by the dimension d of the underlying shape of the finite element space. As we shall see below, indices can be a powerful tool when used to define forms in tensor notation.

An `Index` can also be *fixed*, meaning that the value of the index remains constant:

```
i = Index(0)
```

5.3.10 Built-ins

FFC declares a set of built-in variables and constructors for convenience, as outlined below.

Predefined indices

FFC automatically declares a sequence of free indices for convenience: `i`, `j`, `k`, `l`, `m`, `n`. Note however that a user is free to declare new indices with other names or even reuse these variables for other things than indices.

`Identity`

The data type `Identity` represents an $n \times n$ unit matrix of given size n. An `Identity` is declared by specifying the dimension n:

```
I = Identity(n)
```

`MeshSize`

The function `MeshSize` is a predefined `Function` that may be used to represent the size of the mesh:

```
h = MeshSize(shape)
```

Note that it is the responsibility of the user (or the system for which the code is generated) to map this function to a function (coefficient) that interpolates the mesh size onto piecewise constants.

`FacetNormal`

The function `FacetNormal` is a predefined `Function` that may be used to represent the unit normals of mesh facets.

```
n = FacetNormal(shape)
```

Note that it is the responsibility of the user (or the system for which the code is generated) to map this function to a function (coefficient) that interpolates the facet normals onto vector-valued piecewise constants.

5.4 Scalar operators

The basic operators used to define a form are scalar addition, subtraction and multiplication. Note the absence of division which is intentionally left out (but is supplied for `Function`s, see below).

5.4.1 Scalar addition: +

Scalar addition is supported for all scalar-valued basic data types, thus including `BasisFunction`, `Function`, `Constant` and expressions involving these data types.

In addition, unary plus is supported for all basic data types.

5.4.2 Scalar subtraction: -

Scalar subtraction is supported for all scalar-valued basic data types, thus including `BasisFunction`, `Function`, `Constant` and expressions involving these data types.

In addition, unary minus is supported for all basic data types.

5.4.3 Scalar multiplication: *

Scalar multiplication is supported for all scalar-valued basic data types, thus including `BasisFunction`, `Function`, `Constant` and expressions involving these data types.

5.4.4 Scalar division: /

Division is not allowed for `BasisFunction`s (and thus not for `TestFunction`s and `TrialFunction`s) in the definition of a form. This is because division by a `BasisFunction` in the definition of a form does not result in a valid multilinear form, since a multilinear form must be linear in each of its arguments.

However, division is allowed for `Function`s and is applied to the coefficients of its nodal basis expansion. Thus `1/f` for a `Function f` corresponds to the operation

$$1/f \approx \sum_i (1/f_i)\,\phi_i. \tag{5.4}$$

See also Section 5.9.

5.5 Vector operators

Vectors are defined in the form language using Python's built-in `list` type. This means that all list operations such as slicing, list comprehension etc. are supported. There is one exception to this rule, namely vector-valued `BasisFunction`s and `Function`s, which are not `list`s (but can be made into `list`s using the operator `vec` discussed below). The operators listed below support all objects which are logically vectors, thus including both Python `list`s and vector-valued expressions.

5.5.1 Component access: v[i]

Brackets `[]` are used to pick a given component of a logically vector-valued expression. Thus, if `v` is a vector-valued expression, then `v[0]` represents a function corresponding to the first component of (the values of) `v`. Similarly, if `i` is an `Index` (free or fixed), then `v[i]` represents a function corresponding to component `i` of (the values of) `v`.

5.5.2 Inner product: `dot(v, w)`

The operator `dot` accepts as arguments two logically vector-valued expressions and returns the inner product (dot product) of the two vectors:

$$\texttt{dot(v, w)} \leftrightarrow v \cdot w = \sum_{i=0}^{n-1} v_i w_i. \tag{5.5}$$

Note that this operator is only defined for vectors of equal length.

5.5.3 Vector product: `cross(v, w)`

The operator `cross` accepts as arguments two logically vector-valued expressions and returns a vector which is the cross product (vector product) of the two vectors:

$$\texttt{cross(v, w)} \leftrightarrow v \times w = (v_1 w_2 - v_2 w_1, v_2 w_0 - v_0 w_2, v_0 w_1 - v_1 w_0). \tag{5.6}$$

Note that this operator is only defined for vectors of length three.

5.5.4 Matrix product: `mult(v, w)`

The operator `mult` accepts as arguments two matrices (or more generally, tensors) and returns the matrix (tensor) product.

5.5.5 Transpose: `transp(v)`

The operator `transp` accepts as argument a matrix and returns the transpose of the given matrix:

$$\texttt{transp(v)[i][j]} \leftrightarrow (v^\top)_{ij} = v_{ji}. \tag{5.7}$$

5.5.6 Trace: `trace(v)`

The operator `trace` accepts as argument a square matrix `v` and returns its trace, that is, the sum of its diagonal elements:

$$\texttt{trace}(\text{v}) \leftrightarrow \text{trace}(v) = \sum_{i=0}^{n-1} v_{ii}. \tag{5.8}$$

5.5.7 Vector length: `len(v)`

The operator `len` accepts as argument a logically vector-valued expression and returns its length (the number of vector components).

5.5.8 Rank: `rank(v)`

The operator `rank` returns the rank of the given argument. The rank of an expression is defined as the number of times the operator `[]` can be applied to the expression before a scalar is obtained. Thus, the rank of a scalar is zero, the rank of a vector is one and the rank of a matrix is two.

5.5.9 Vectorization: `vec(v)`

The operator `vec` is used to create a Python `list` object from a logically vector-valued expression. This operator has no effect on expressions which are already `list`s. Thus, if `v` is a vector-valued `BasisFunction`, then `vec(v)` returns a list of the components of `v`. This can be used to define forms in terms of standard Python `list` operators or Python NumPy `array` operators.

The operator `vec` does not have to be used if the form is defined only in terms of the basic operators of the form language.

5.6 Differential operators

5.6.1 Scalar partial derivative: `D(v, i)`

The basic differential operator is the scalar partial derivative `D`. This differential operator accepts as arguments a scalar or logically vector-valued expression `v` together with a coordinate direction `i` and returns the partial derivative of the expression in the given coordinate direction:

$$\texttt{D(v, i)} \leftrightarrow \frac{\partial v}{\partial x_i}. \tag{5.9}$$

Alternatively, the member function `dx` can be used. For `v` an expression, the two expressions `D(v, i)` and `v.dx(i)` are equivalent, but note that only the operator `D` works on vector-valued expressions that are defined in terms of Python lists.

5.6.2 Gradient: `grad(v)`

The operator `grad` accepts as argument an expression `v` and returns its gradient. If `v` is scalar, the result is a vector containing the partial derivatives in the coordinate directions:

$$\texttt{grad(v)} \leftrightarrow \text{grad(v)} = \nabla v = (\frac{\partial v}{\partial x_0}, \frac{\partial v}{\partial x_1}, \ldots, \frac{\partial v}{\partial x_{d-1}}). \tag{5.10}$$

If `v` is logically vector-valued, the result is a matrix with rows given by the gradients of each component:

$$\texttt{grad(v)}[i][j] \leftrightarrow (\text{grad(v)})_{ij} = (\nabla v)_{ij} = \frac{\partial v_i}{\partial x_j}. \tag{5.11}$$

Thus, if `v` is scalar-valued, then `grad(grad(v))` returns the Hessian of `v`, and if `v` is vector-valued, then `grad(v)` is the Jacobian of `v`.

5.6.3 Divergence: `div(v)`

The operator `div` accepts as argument a logically vector-valued expression and returns its divergence:

$$\texttt{div(v)} \leftrightarrow \operatorname{div} v = \nabla \cdot v = \sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i}. \tag{5.12}$$

Note that the length n of the vector `v` must be equal to the dimension d of the underlying shape of the `FiniteElement` defining the function space for `v`.

5.6.4 Curl: `curl(v)`

The operator `curl` accepts as argument a logically vector-valued expression and returns its curl:

$$\texttt{curl(v)} \leftrightarrow \operatorname{curl} v = \nabla \times v = (\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}). \tag{5.13}$$

Note that this operator is only defined for vectors of length three.

Alternatively, the name `rot` can be used for this operator.

5.7 Integrals

Each term of a valid form expression must be a scalar-valued expression integrated exactly once. Integrals are expressed through multiplication with a measure, representing either an integral over the interior of the domain Ω (cell integral), the boundary $\partial\Omega$ of Ω (exterior facet integral) or the set of interior facets (interior facet integral).

5.7.1 Cell integrals: `*dx`

A measure for integration over the interior of Ω is created as follows:

```
dx = Integral("cell")
```

For convenience, **FFC** automatically declares the measure `dx` which can be used to define cell integrals. If `v` is a scalar-valued expression, then the integral of `v` over the interior of Ω is written as `v*dx`.

5.7.2 Exterior facet integrals: `*ds`

A measure for integration over the boundary of Ω is created as follows:

```
ds = Integral("exterior facet")
```

For convenience, **FFC** automatically declares the measure `ds` which can be used to define cell integrals. If `v` is a scalar-valued expression, then the integral of `v` over the boundary of Ω is written as `v*ds`.

5.7.3 Interior facet integrals: `*dS`

A measure for integration over the set of interior facets of Ω is created as follows:

```
dS = Integral("interior facet")
```

For convenience, **FFC** automatically declares the measure `dS` which can be used to define cell integrals. If `v` is a scalar-valued expression, then the integral of `v` over the interior facets of Ω is written as `v*dS`.

5.7.4 Integrals over subsets

Integrals over multiple disjoint subdomains of Ω may be defined by specifying an additional argument for the number of the subdomain associated with each integral. The different measures may then be combined to express a form as a sum of integrals over the different subdomains.

```
dx0 = Integral("cell", 0)
dx1 = Integral("cell", 1)

ds0 = Integral("exterior facet", 0)
ds1 = Integral("exterior facet", 1)
ds2 = Integral("exterior facet", 2)

dS0 = Integral("interior facet", 0)

a = ...*dx0 + ...*dx1 + ...*ds0 + ...*ds1 + ...*ds2 + ...*dS0
```

5.8 DG operators

FFC provides operators for implementation of discontinuous Galerkin methods. These include the evaluation of the jump and average of a function (or in general an expression) over the interior facets (edges or faces) of a mesh.

5.8.1 Restriction: `v('+')` and `v('-')`

When integrating over interior facets (`*dS`), one may restrict expressions to the positive or negative side of the facet:

```
element = FiniteElement("Discontinuous Lagrange",
                        "tetrahedron", 0)
```

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

f = Function(element)

a = f('+')*dot(grad(v)('+'), grad(u)('-'))*dS
```

Restriction may be applied to functions of any finite element space but will only have effect when applied to expressions that are discontinuous across facets.

5.8.2 Jump: `jump(v)`

The operator `jump` may be used to express the jump of a function across a common facet of two cells. Two versions of the `jump` operator are provided.

If called with only one argument, then the `jump` operator evaluates to the difference between the restrictions of the given expression on the positive and negative sides of the facet:

$$\texttt{jump(v)} \leftrightarrow [\![v]\!] = v^+ - v^-. \tag{5.14}$$

If the expression `v` is scalar, then `jump(v)` will also be scalar, and if `v` is vector-valued, then `jump(v)` will also be vector-valued.

If called with two arguments, `jump(v, n)` evaluates to the jump in `v` weighted by `n`. Typically, `n` will be chosen to represent the unit outward normal of the facet (as seen from each of the two neighboring cells). If `v` is scalar, then `jump(v, n)` is given by

$$\texttt{jump(v, n)} \leftrightarrow [\![v]\!]_n = v^+ n^+ + v^- n^-. \tag{5.15}$$

If `v` is vector-valued, then `jump(v, n)` is given by

$$\texttt{jump(v, n)} \leftrightarrow [\![v]\!]_n = v^+ \cdot n^+ + v^- \cdot n^-. \tag{5.16}$$

Thus, if the expression `v` is scalar, then `jump(v, n)` will be vector-valued, and if `v` is vector-valued, then `jump(v, n)` will be scalar.

5.8.3 Average: `avg(v)`

The operator `avg` may be used to express the average of a function across a common facet of two cells:

$$\text{avg(v)} \leftrightarrow \langle v \rangle = \frac{1}{2}(v^+ + v^-). \tag{5.17}$$

If the expression `v` is scalar, then `avg(v)` will also be scalar, and if `v` is vector-valued, then `avg(v)` will also be vector-valued.

5.9 Special operators

FFC provides a set of special operators for taking the inverse, absolute value and square root of an expression. These operators are interpreted in a special way and should be used with care. Firstly, the operators are only valid on monomial expressions, that is, expressions that consist of only one term. Secondly, the operators are applied directly to the *coefficients* of the basis function expansion of the expression on which the operators are applied. Thus, if $v = \sum_i v_i \phi_i$, then $\text{op}(v)$ is evaluated by

$$\text{op}(v) = \sum_i \text{op}(v_i)\phi_i. \tag{5.18}$$

5.9.1 Inverse: `1/v`

The inverse of a monomial expression (for example a product of one or more functions) may be evaluated (in the sense described above) as follows:

```
w = 1/v
```

5.9.2 Absolute value: `abs(v)`

The absolute value of a monomial expression (for example a product of one or more functions) may be evaluated (in the sense described above) as follows:

```
w = abs(v)
```

5.9.3 Square root: `sqrt(v)`

The square root of a monomial expression (for example a product of one or more functions) may be evaluated (in the sense described above) as follows:

```
w = sqrt(v)
```

5.9.4 Combining operators

The special operators may applied successively and repeatedly on any monomial expression. Thus, the following expression is valid:

```
v = Function(element)
w = sqrt(abs(1/v))
```

5.10 Index notation

FFC supports index notation, which is often a convenient way to express forms. The basic principle of index notation is that summation is implicit over indices repeated twice in each term of an expression. The following

examples illustrate the index notation, assuming that each of the variables `i` and `j` have been declared as a free `Index`:

$$\texttt{v[i]*w[i]} \quad \leftrightarrow \quad \sum_{i=0}^{n-1} v_i w_i, \tag{5.19}$$

$$\texttt{D(v, i)*D(w, i)} \quad \leftrightarrow \quad \sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i}\frac{\partial w}{\partial x_i} = \nabla v \cdot \nabla w, \tag{5.20}$$

$$\texttt{D(v[i], i)} \quad \leftrightarrow \quad \sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i} = \nabla \cdot v, \tag{5.21}$$

$$\texttt{D(v[i], j)*D(w[i], j)} \quad \leftrightarrow \quad \sum_{i=0}^{n-1}\sum_{j=0}^{d-1} \frac{\partial v_i}{\partial x_j}\frac{\partial w_i}{\partial x_j}. \tag{5.22}$$

Index notation is used internally by **FFC** to represent multilinear forms and **FFC** will try to simplify forms by replacing sums with index expressions.

5.11 User-defined operators

A user may define new operators, using standard Python syntax. As an example, consider the strain-rate operator ϵ of linear elasticity, defined by

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top). \tag{5.23}$$

This operator can be implemented as a function using the Python `def` keyword:

```
def epsilon(v):
    return 0.5*(grad(v) + transp(grad(v)))
```

Alternatively, using the shorthand `lambda` notation, the strain operator may be defined as follows:

```
epsilon = lambda v: 0.5*(grad(v) + transp(grad(v)))
```

Chapter 6

Examples

The following examples illustrate basic usage of the form language for the definition of a collection of standard multilinear forms. We assume that `dx` has been declared as an integral over the interior of Ω and that both `i` and `j` have been declared as a free `Index`.

The examples presented below can all be found in the subdirectory `src/demo` of the **FFC** source tree together with numerous other examples.

6.1 The mass matrix

As a first example, consider the bilinear form corresponding to a mass matrix,

$$a(v, u) = \int_\Omega v\, u \,\mathrm{d}x, \tag{6.1}$$

which can be implemented in **FFC** as follows:

```
element = FiniteElement("Lagrange", "triangle", 1)

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

```
a = v*u*dx
```

This example is implemented in the file `Mass.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.2 Poisson's equation

The bilinear and linear forms form for Poisson's equation,

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x, \tag{6.2}$$

$$L(v) = \int_\Omega v \, f \, \mathrm{d}x, \tag{6.3}$$

can be implemented as follows:

```
element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

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

Alternatively, index notation can be used to express the scalar product:

```
a = D(v, i)*D(u, i)*dx
```

This example is implemented in the file `Poisson.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.3 Vector-valued Poisson

The bilinear and linear forms for a system of (independent) Poisson equations,

$$a(v, u) = \int_\Omega \nabla v : \nabla u \, \mathrm{d}x, \tag{6.4}$$

$$L(v) = \int_\Omega v \cdot f \, \mathrm{d}x, \tag{6.5}$$

with v, u and f vector-valued can be implemented as follows:

```
element = VectorElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)

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

Alternatively, index notation may be used:

```
a = D(v[i], j)*D(u[i], j)*dx
L = v[i]*f[i]*dx
```

This example is implemented in the file `PoissonSystem.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.4 The strain-strain term of linear elasticity

The strain-strain term of linear elasticity,

$$a(v, u) = \int_\Omega \epsilon(v) : \epsilon(u) \, \mathrm{d}x, \tag{6.6}$$

where

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top) \tag{6.7}$$

can be implemented as follows:

```
element = VectorElement("Lagrange", "tetrahedron", 1)

v = TestFunction(element)
u = TrialFunction(element)

def epsilon(v):
    return 0.5*(grad(v) + transp(grad(v)))

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

Alternatively, index notation can be used to define the form:

```
a = 0.25*(D(v[i], j) + D(v[j], i))* \
         (D(u[i], j) + D(u[j], i))*dx
```

This example is implemented in the file `Elasticity.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.5 The nonlinear term of Navier–Stokes

The bilinear form for fixed-point iteration on the nonlinear term of the incompressible Navier–Stokes equations,

$$a(v, u) = \int_\Omega v \cdot ((w \cdot \nabla)u) \, \mathrm{d}x, \tag{6.8}$$

with w the frozen velocity from a previous iteration, can be conveniently implemented using index notation as follows:

```
element = FiniteElement("Vector Lagrange", "tetrahedron", 1)

v = TestFunction(element)
u = TrialFunction(element)
w = Function(element)

a = v[i]*w[j]*D(u[i], j)*dx
```

This example is implemented in the file `NavierStokes.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.6 The heat equation

Discretizing the heat equation,

$$\dot{u} - \nabla \cdot (c \nabla u) = f, \tag{6.9}$$

in time using the dG(0) method (backward Euler), we obtain the following variational problem for the discrete solution $u_h = u_h(x, t)$: Find $u_h^n = u_h(\cdot, t_n)$ with $u_h^{n-1} = u_h(\cdot, t_{n-1})$ given such that

$$\frac{1}{k_n} \int_\Omega v \, (u_h^n - u_h^{n-1}) \, \mathrm{d}x + \int_\Omega c \, \nabla v \cdot \nabla u_h^n \, \mathrm{d}x = \int_\Omega v \, f^n \, \mathrm{d}x \tag{6.10}$$

for all test functions v, where $k = t_n - t_{n-1}$ denotes the time step . In the example below, we implement this variational problem with piecewise linear test and trial functions, but other choices are possible (just choose another finite element).

Rewriting the variational problem in the standard form $a(v, u_h) = L(v)$ for all v, we obtain the following pair of bilinear and linear forms:

$$a(v, u_h^n) = \int_\Omega v \, u_h^n \, \mathrm{d}x + k_n \int_\Omega c \, \nabla v \cdot \nabla u_h^n \, \mathrm{d}x, \tag{6.11}$$

$$L(v) = \int_\Omega v \, u_h^{n-1} \, \mathrm{d}x + k_n \int_\Omega v \, f^n \, \mathrm{d}x, \tag{6.12}$$

which can be implemented as follows:

```
element = FiniteElement("Lagrange", "triangle", 1)

v  = TestFunction(element) # Test function
u1 = TrialFunction(element) # Value at t_n
u0 = Function(element)      # Value at t_n-1
c  = Function(element)      # Heat conductivity
f  = Function(element)      # Heat source
k  = Constant()             # Time step

a = v*u1*dx + k*c*dot(grad(v), grad(u1))*dx
L = v*u0*dx + k*v*f*dx
```

6.7 Mixed formulation of Stokes

To solve Stokes' equations,

$$-\Delta u + \nabla p = f, \tag{6.13}$$
$$\nabla \cdot u = 0, \tag{6.14}$$

we write the variational problem in standard form $a(v, u) = L(v)$ for all v to obtain the following pair of bilinear and linear forms:

$$a((v, q), (u, p)) = \int_\Omega \nabla v : \nabla u - (\nabla \cdot v)\, p + q\, (\nabla \cdot u)\, \mathrm{d}x, \tag{6.15}$$
$$L((v, q)) = \int_\Omega v \cdot f\, \mathrm{d}x. \tag{6.16}$$

Using a mixed formulation with Taylor-Hood elements, this can be implemented as follows:

```
P2 = FiniteElement("Vector Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = P2 + P1

(v, q) = TestFunctions(TH)
```

```
(u, p) = TrialFunctions(TH)

f = Function(P2)

a = (dot(grad(v), grad(u)) - div(v)*P + q*div(u))*dx
L = dot(v, f)*dx
```

This example is implemented in the file `Heat.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.8 Mixed formulation of Poisson

We next consider the following formulation of Poisson's equation as a pair of first order equations for $\sigma \in H(\text{div})$ and $u \in L_2$:

$$\sigma + \nabla u = 0, \tag{6.17}$$
$$\nabla \cdot \sigma = f. \tag{6.18}$$

We multiply the two equations by a pair of test functions τ and w and integrate by parts to obtain the following variational problem: Find $(\sigma, u) \in V = H(\text{div}) \times L_2$ such that

$$a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall\, (\tau, w) \in V, \tag{6.19}$$

where

$$a((\tau, w), (\sigma, u)) = \int_\Omega \tau \cdot \sigma - \nabla \cdot \tau\, u + w \nabla \cdot \sigma \,\mathrm{d}x, \tag{6.20}$$
$$L((\tau, w)) = \int_\Omega w \cdot f \,\mathrm{d}x. \tag{6.21}$$

We may implement the corresponding forms in the **FFC** form language using first order BDM $H(\text{div})$-conforming elements for σ and piecewise constant L_2-conforming elements for u as follows:

```
BDM1 = FiniteElement("Brezzi-Douglas-Marini", "triangle", 1)
DG0  = FiniteElement("Discontinuous Lagrange", "triangle", 0)

element = BDM1 + DG0

(tau, w) = TestFunctions(element)
(sigma, u) = TrialFunctions(element)

f = Function(DG0)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx
L = w*f*dx
```

This example is implemented in the file `MixedPoisson.form` in the collection of demonstration forms included with the **FFC** source distribution.

6.9 Poisson's equation with DG elements

We consider again Poisson's equation, but now in an (interior penalty) discontinuous Galerkin formulation: Find $u \in V = L_2$ such that

$$a(v, u) = L(v) \quad \forall v \in V,$$

where

$$\begin{aligned} a(v,u) &= \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x \\ &\quad + \sum_S \int_S -\langle \nabla v \rangle \cdot [\![u]\!]_n - [\![v]\!]_n \cdot \langle \nabla u \rangle + (\alpha/h)[\![v]\!]_n \cdot [\![u]\!]_n \,\mathrm{d}S \\ &\quad + \int_{\partial\Omega} -\nabla v \cdot [\![u]\!]_n - [\![v]\!]_n \cdot \nabla u + (\gamma/h) v u \,\mathrm{d}s \\ L(v) &= \int_\Omega v f \,\mathrm{d}x + \int_{\partial\Omega} v g \,\mathrm{d}s. \end{aligned} \tag{6.22}$$

The corresponding finite element variational problem for discontinuous first order elements may be implemented as follows:

```
DG1 = FiniteElement("Discontinuous Lagrange", "triangle", 1)

v = TestFunction(DG1)
u = TrialFunction(DG1)

f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")

a = dot(grad(v), grad(u))*dx \
  - dot(avg(grad(v)), jump(u, n))*dS \
  - dot(jump(v, n), avg(grad(u)))*dS \
  + alpha/h('+')*dot(jump(v, n), jump(u, n))*dS \
  - dot(grad(v), jump(u, n))*ds \
  - dot(jump(v, n),\ grad(u))*ds \
  + gamma/h*v*u*ds
L = v*f*dx + v*g*ds
```

This example is implemented in the file `PoissonDG.form` in the collection of demonstration forms included with the **FFC** source distribution.

Bibliography

[1] *Mercurial*, 2006. http://www.selenic.com/mercurial/.

[2] M. S. Alnæs, H. P. Langtangen, A. Logg, K.-A. M. dal, and O. Skavhaug, *UFC*, 2007. URL: http://www.fenics.org/ufc/.

[3] ——, *UFC specification and user manual*, 2007. URL: http://www.fenics.org/ufc/.

[4] J. Hoffman, J. Jansson, A. Logg, and G. N. Wells, *DOLFIN*, 2006. URL: http://www.fenics.org/dolfin/.

[5] R. C. Kirby and A. Logg, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), pp. 417–444.

[6] ——, *Efficient compilation of a class of variational forms*, ACM Transactions on Mathematical Software, 33 (2007).

Appendix A

Reference cells

The definition of reference cells used in **FFC** follows the UFC specification. [2, 3]

The following five reference cells are covered by the UFC specification: the reference *interval*, the reference *triangle*, the reference *quadrilateral*, the reference *tetrahedron* and the reference *hexahedron*.

Reference cell	Dimension	#Vertices	#Facets
The reference interval	1	2	2
The reference triangle	2	3	3
The reference quadrilateral	2	4	4
The reference tetrahedron	3	4	4
The reference hexahedron	3	8	6

Table A.1: Reference cells covered by the UFC specification.

The UFC specification assumes that each cell in a finite element mesh is always isomorphic to one of the reference cells.

A.1 The reference interval

The reference interval is shown in Figure A.1 and is defined by its two vertices with coordinates as specified in Table A.2.

Figure A.1: The reference interval.

Vertex	Coordinate
v_0	$x = 0$
v_1	$x = 1$

Table A.2: Vertex coordinates of the reference interval.

A.2 The reference triangle

The reference triangle is shown in Figure A.2 and is defined by its three vertices with coordinates as specified in Table A.3.

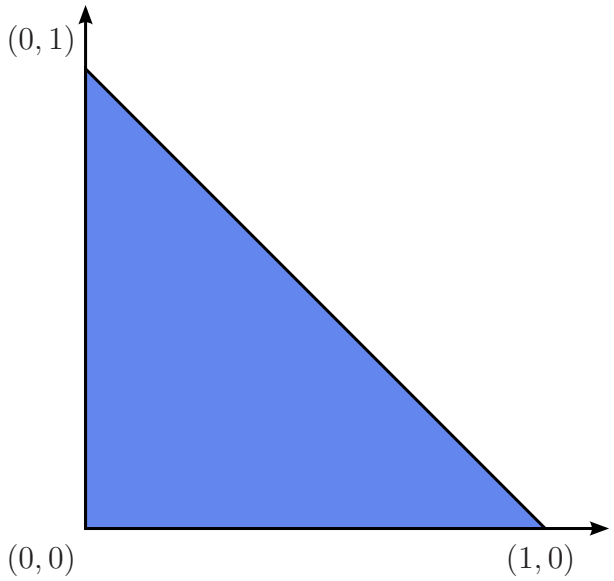

Figure A.2: The reference triangle.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (0, 1)$

Table A.3: Vertex coordinates of the reference triangle.

A.3 The reference quadrilateral

The reference quadrilateral is shown in Figure A.3 and is defined by its four vertices with coordinates as specified in Table A.4.

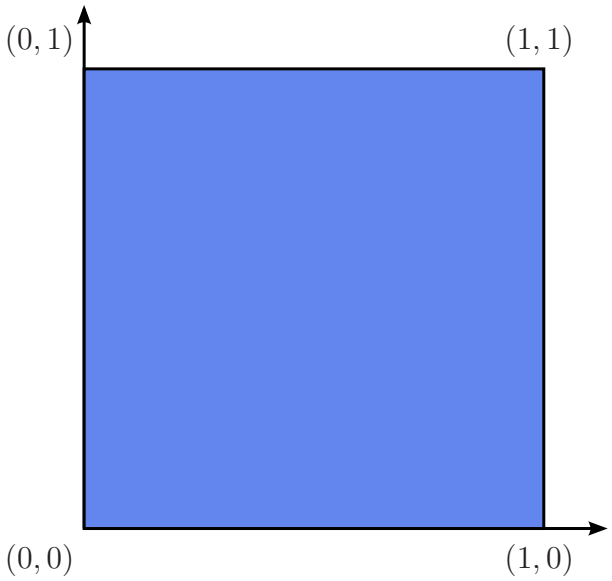

Figure A.3: The reference quadrilateral.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (1, 1)$
v_3	$x = (0, 1)$

Table A.4: Vertex coordinates of the reference quadrilateral.

A.4 The reference tetrahedron

The reference tetrahedron is shown in Figure A.4 and is defined by its four vertices with coordinates as specified in Table A.5.

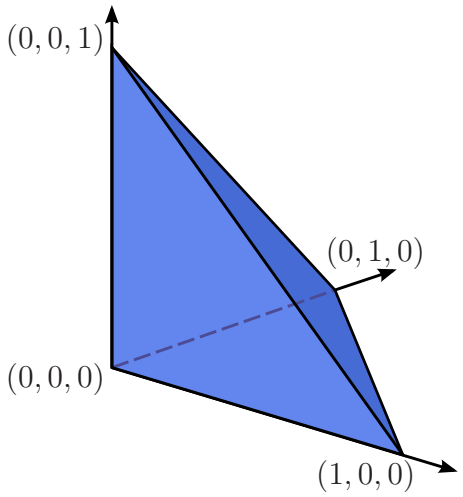

Figure A.4: The reference tetrahedron.

Vertex	Coordinate
v_0	$x = (0, 0, 0)$
v_1	$x = (1, 0, 0)$
v_2	$x = (0, 1, 0)$
v_3	$x = (0, 0, 1)$

Table A.5: Vertex coordinates of the reference tetrahedron.

A.5 The reference hexahedron

The reference hexahedron is shown in Figure A.5 and is defined by its eight vertices with coordinates as specified in Table A.6.

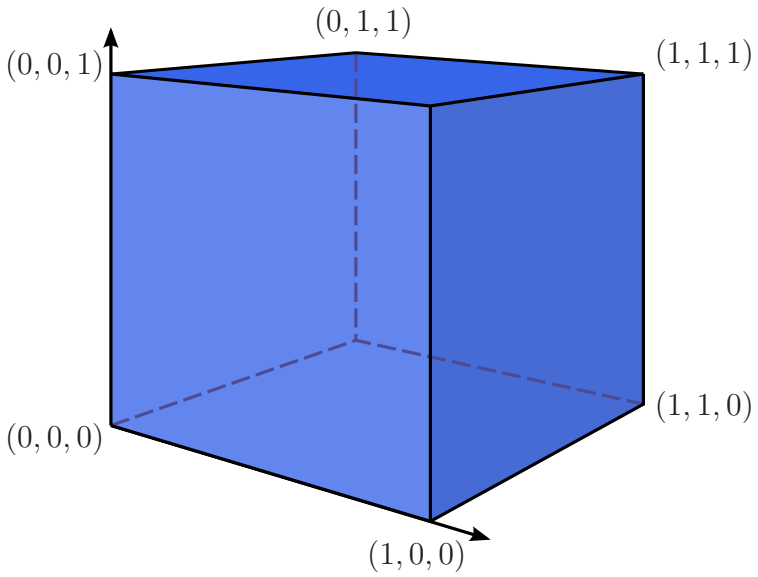

Figure A.5: The reference hexahedron.

Vertex	Coordinate	Vertex	Coordinate
v_0	$x = (0, 0, 0)$	v_4	$x = (0, 0, 1)$
v_1	$x = (1, 0, 0)$	v_5	$x = (1, 0, 1)$
v_2	$x = (1, 1, 0)$	v_6	$x = (1, 1, 1)$
v_3	$x = (0, 1, 0)$	v_7	$x = (0, 1, 1)$

Table A.6: Vertex coordinates of the reference hexahedron.

Appendix B

Numbering of mesh entities

The numbering of mesh entities used in **FFC** follows the UFC specification. [2, 3]

The UFC specification dictates a certain numbering of the vertices, edges etc. of the cells of a finite element mesh. First, an *ad hoc* numbering is picked for the vertices of each cell. Then, the remaining entities are ordered based on a simple rule, as described in detail below.

B.1 Basic concepts

The topological entities of a cell (or mesh) are referred to as *mesh entities*. A mesh entity can be identified by a pair (d, i), where d is the topological dimension of the mesh entity and i is a unique index of the mesh entity. Mesh entities are numbered within each topological dimension from 0 to $n_d - 1$, where n_d is the number of mesh entities of topological dimension d.

For convenience, mesh entities of topological dimension 0 are referred to as *vertices*, entities of dimension 1 as *edges*, entities of dimension 2 as *faces*, entities of *codimension* 1 as *facets* and entities of codimension 0 as *cells*. These concepts are summarized in Table B.1.

Thus, the vertices of a tetrahedron are identified as $v_0 = (0, 0)$, $v_1 = (0, 1)$ and $v_2 = (0, 2)$, the edges are $e_0 = (1, 0)$, $e_1 = (1, 1)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$, $e_4 = (1, 4)$ and $e_5 = (1, 5)$, the faces (facets) are $f_0 = (2, 0)$, $f_1 = (2, 1)$, $f_2 = (2, 2)$ and $f_3 = (2, 3)$, and the cell itself is $c_0 = (3, 0)$.

Entity	Dimension	Codimension
Vertex	0	–
Edge	1	–
Face	2	–
Facet	–	1
Cell	–	0

Table B.1: Named mesh entities.

B.2 Numbering of vertices

For simplicial cells (intervals, triangles and tetrahedra) of a finite element mesh, the vertices are numbered locally based on the corresponding global vertex numbers. In particular, a tuple of increasing local vertex numbers corresponds to a tuple of increasing global vertex numbers. This is illustrated in Figure B.1 for a mesh consisting of two triangles.

For non-simplicial cells (quadrilaterals and hexahedra), the numbering is arbitrary, as long as each cell is isomorphic to the corresponding reference cell by matching each vertex with the corresponding vertex in the reference cell. This is illustrated in Figure B.2 for a mesh consisting of two quadrilaterals.

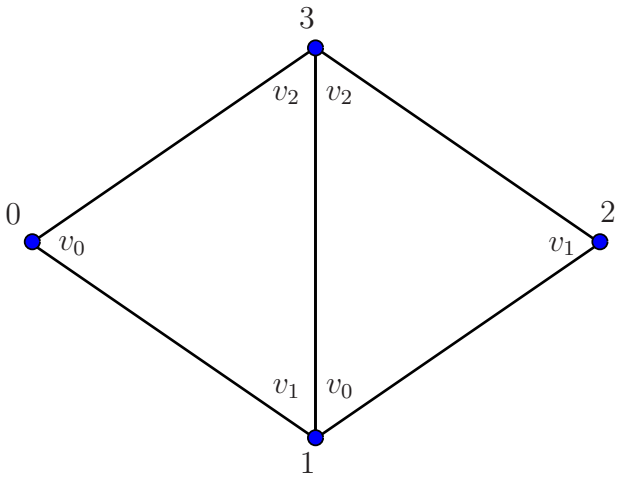

Figure B.1: The vertices of a simplicial mesh are numbered locally based on the corresponding global vertex numbers.

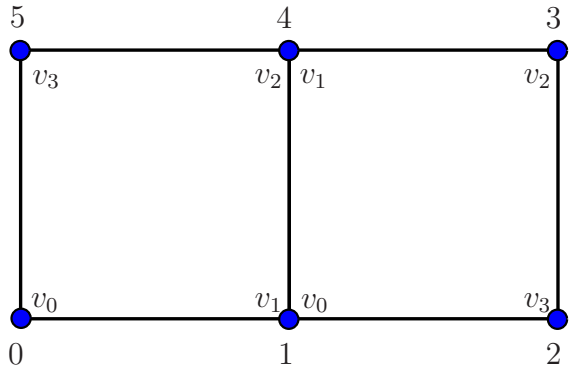

Figure B.2: The local numbering of vertices of a non-simplicial mesh is arbitrary, as long as each cell is isomorphic to the reference cell by matching each vertex to the corresponding vertex of the reference cell.

B.3 Numbering of other mesh entities

When the vertices have been numbered, the remaining mesh entities are numbered within each topological dimension based on a *lexicographical ordering* of the corresponding ordered tuples of *non-incident vertices*.

As an illustration, consider the numbering of edges (the mesh entities of topological dimension one) on the reference triangle in Figure B.3. To number the edges of the reference triangle, we identify for each edge the corresponding non-incident vertices. For each edge, there is only one such vertex (the vertex opposite to the edge). We thus identify the three edges in the reference triangle with the tuples (v_0), (v_1) and (v_2). The first of these is edge e_0 between vertices v_1 and v_2 opposite to vertex v_0, the second is edge e_1 between vertices v_0 and v_2 opposite to vertex v_1, and the third is edge e_2 between vertices v_0 and v_1 opposite to vertex v_2.

Similarly, we identify the six edges of the reference tetrahedron with the corresponding non-incident tuples (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3) and (v_2, v_3). The first of these is edge e_0 between vertices v_2 and v_3 opposite to vertices v_0 and v_1 as shown in Figure B.4.

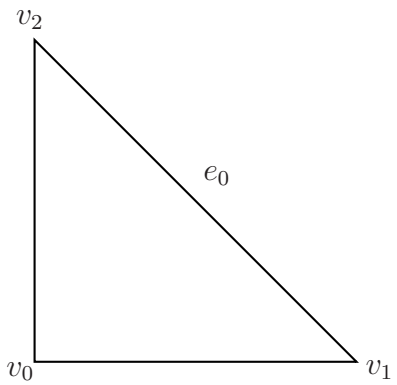

Figure B.3: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertex v_0.

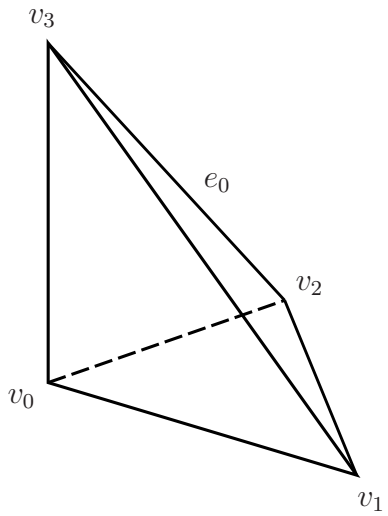

Figure B.4: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertices v_0 and v_1.

B.3.1 Relative ordering

The relative ordering of mesh entities with respect to other incident mesh entities follows by sorting the entities by their (global) indices. Thus, the pair of vertices incident to the first edge e_0 of a triangular cell is (v_1, v_2), not (v_2, v_1). Similarly, the first face f_0 of a tetrahedral cell is incident to vertices (v_1, v_2, v_3).

For simplicial cells, the relative ordering in combination with the convention of numbering the vertices locally based on global vertex indices means that two incident cells will always agree on the orientation of incident subsimplices. Thus, two incident triangles will agree on the orientation of the common edge and two incident tetrahedra will agree on the orientation of the common edge(s) and the orientation of the common face (if any). This is illustrated in Figure B.5 for two incident triangles sharing a common edge.

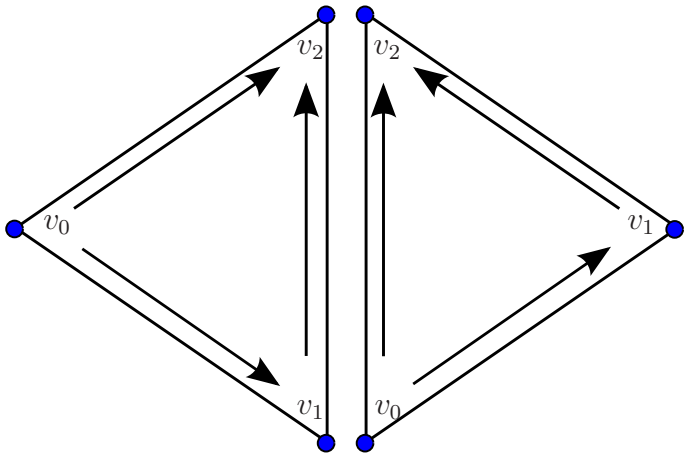

Figure B.5: Two incident triangles will always agree on the orientation of the common edge.

B.3.2 Limitations

The UFC specification is only concerned with the ordering of mesh entities with respect to entities of larger topological dimension. In other words, the UFC specification is only concerned with the ordering of incidence relations of the class $d - d'$ where $d > d'$. For example, the UFC specification is not concerned with the ordering of incidence relations of the class $0 - 1$, that is, the ordering of edges incident to vertices.

B.4 Numbering schemes for reference cells

The numbering scheme is demonstrated below for cells isomorphic to each of the five reference cells.

B.4.1 Numbering for intervals

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1)
$v_1 = (0, 1)$	(v_1)	(v_0)
$c_0 = (1, 0)$	(v_0, v_1)	$\emptyset$

Table B.2: Numbering of mesh entities on intervals.

B.4.2 Numbering for triangular cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1)
$e_0 = (1, 0)$	(v_1, v_2)	(v_0)
$e_1 = (1, 1)$	(v_0, v_2)	(v_1)
$e_2 = (1, 2)$	(v_0, v_1)	(v_2)
$c_0 = (2, 0)$	(v_0, v_1, v_2)	$\emptyset$

Table B.3: Numbering of mesh entities on triangular cells.

B.4.3 Numbering for quadrilateral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_2)	(v_0, v_3)
$e_2 = (1, 2)$	(v_0, v_3)	(v_1, v_2)
$e_3 = (1, 3)$	(v_0, v_1)	(v_2, v_3)
$c_0 = (2, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Table B.4: Numbering of mesh entities on quadrilateral cells.

B.4.4 Numbering for tetrahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_3)	(v_0, v_2)
$e_2 = (1, 2)$	(v_1, v_2)	(v_0, v_3)
$e_3 = (1, 3)$	(v_0, v_3)	(v_1, v_2)
$e_4 = (1, 4)$	(v_0, v_2)	(v_1, v_3)
$e_5 = (1, 5)$	(v_0, v_1)	(v_2, v_3)
$f_0 = (2, 0)$	(v_1, v_2, v_3)	(v_0)
$f_1 = (2, 1)$	(v_0, v_2, v_3)	(v_1)
$f_2 = (2, 2)$	(v_0, v_1, v_3)	(v_2)
$f_3 = (2, 3)$	(v_0, v_1, v_2)	(v_3)
$c_0 = (3, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Table B.5: Numbering of mesh entities on tetrahedral cells.

B.4.5 Numbering for hexahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	$(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_1 = (0, 1)$	(v_1)	$(v_0, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_2 = (0, 2)$	(v_2)	$(v_0, v_1, v_3, v_4, v_5, v_6, v_7)$
$v_3 = (0, 3)$	(v_3)	$(v_0, v_1, v_2, v_4, v_5, v_6, v_7)$
$v_4 = (0, 4)$	(v_4)	$(v_0, v_1, v_2, v_3, v_5, v_6, v_7)$
$v_5 = (0, 5)$	(v_5)	$(v_0, v_1, v_2, v_3, v_4, v_6, v_7)$
$v_6 = (0, 6)$	(v_6)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_7)$
$v_7 = (0, 7)$	(v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
$e_0 = (1, 0)$	(v_6, v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5)$
$e_1 = (1, 1)$	(v_5, v_6)	$(v_0, v_1, v_2, v_3, v_4, v_7)$
$e_2 = (1, 2)$	(v_4, v_7)	$(v_0, v_1, v_2, v_3, v_5, v_6)$
$e_3 = (1, 3)$	(v_4, v_5)	$(v_0, v_1, v_2, v_3, v_6, v_7)$
$e_4 = (1, 4)$	(v_3, v_7)	$(v_0, v_1, v_2, v_4, v_5, v_6)$
$e_5 = (1, 5)$	(v_2, v_6)	$(v_0, v_1, v_3, v_4, v_5, v_7)$
$e_6 = (1, 6)$	(v_2, v_3)	$(v_0, v_1, v_4, v_5, v_6, v_7)$
$e_7 = (1, 7)$	(v_1, v_5)	$(v_0, v_2, v_3, v_4, v_6, v_7)$
$e_8 = (1, 8)$	(v_1, v_2)	$(v_0, v_3, v_4, v_5, v_6, v_7)$
$e_9 = (1, 9)$	(v_0, v_4)	$(v_1, v_2, v_3, v_5, v_6, v_7)$
$e_{10} = (1, 10)$	(v_0, v_3)	$(v_1, v_2, v_4, v_5, v_6, v_7)$
$e_{11} = (1, 11)$	(v_0, v_1)	$(v_2, v_3, v_4, v_5, v_6, v_7)$
$f_0 = (2, 0)$	(v_4, v_5, v_6, v_7)	(v_0, v_1, v_2, v_3)
$f_1 = (2, 1)$	(v_2, v_3, v_6, v_7)	(v_0, v_1, v_4, v_5)
$f_2 = (2, 2)$	(v_1, v_2, v_5, v_6)	(v_0, v_3, v_4, v_7)
$f_3 = (2, 3)$	(v_0, v_3, v_4, v_7)	(v_1, v_2, v_5, v_6)
$f_4 = (2, 4)$	(v_0, v_1, v_4, v_5)	(v_2, v_3, v_6, v_7)
$f_5 = (2, 5)$	(v_0, v_1, v_2, v_3)	(v_4, v_5, v_6, v_7)
$c_0 = (3, 0)$	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$	$\emptyset$

Table B.6: Numbering of mesh entities on hexahedral cells.

Appendix C

Installation

The source code of **FFC** is portable and should work on any system with a standard Python installation. Questions, bug reports and patches concerning the installation should be directed to the **FFC** mailing list at the address

```
ffc-dev@fenics.org
```

FFC must currently be installed directly from source, but Debian (Ubuntu) packages will be available in the future, for **FFC** and other **FEniCS** components.

C.1 Installing from source

C.1.1 Dependencies and requirements

FFC depends on a number of libraries that need to be installed on your system. These libraries include **FIAT** and the Python NumPy module. In addition, you need to have a working Python installation on your system.

Installing Python

FFC is developed for Python 2.5, but should also work with Python 2.3 and 2.4. To check which version of Python you have installed, issue the command `python -V`:

```
# python -V
Python 2.5.1
```

If Python is not installed on your system, it can be downloaded from

```
http://www.python.org/
```

Follow the installation instructions for Python given on the Python web page. For Debian (Ubuntu) users, the package to install is named `python`.

Installing NumPy

In addition to Python itself, **FFC** depends on the Python package NumPy, which is used by **FFC** to process multidimensional arrays (tensors). Python NumPy can be downloaded from

```
http://www.scipy.org/
```

For Debian (Ubuntu) users, the package to install is `python-numpy`.

Installing FIAT

FFC depends on the latest version of **FIAT**, which can be downloaded from

```
http://www.fenics.org/
```

FIAT is used by **FFC** to create and evaluate finite element basis functions and quadrature rules. The installation instructions for **FIAT** are similar to those for **FFC** given in detail below.

C.1.2 Downloading the source code

The latest release of **FFC** can be obtained as a `tar.gz` archive in the download section at

```
http://www.fenics.org/
```

Download the latest release of **FFC**, for example `ffc-x.y.z.tar.gz`, and unpack using the command

```
# tar zxfv ffc-x.y.z.tar.gz
```

This creates a directory `ffc-x.y.z` containing the **FFC** source code.

If you want the very latest version of **FFC**, it can be accessed directly from the development repository through `hg` (Mercurial):

```
# hg clone http://www.fenics.org/hg/ffc
```

This version may contain features not yet present in the latest release, but may also be less stable and even not work at all.

C.1.3 Installing FFC

FFC follows the standard installation procedure for Python packages. Enter the source directory of **FFC** and issue the following command:

```
# python setup.py install
```

This will install the **FFC** Python package in a subdirectory called `ffc` in the default location for user-installed Python packages (usually something like `/usr/lib/python2.5/site-packages`). In addition, the compiler executable `ffc` (a Python script) will be installed in the default directory for user-installed Python scripts (usually in `/usr/bin`).

To see a list of optional parameters to the installation script, type

```
# python setup.py install --help
```

If you don't have root access to the system you are using, you can pass the `--home` option to the installation script to install **FFC** in your home directory:

```
# mkdir ~/local
# python setup.py install --home ~/local
```

This installs the **FFC** package in the directory `~/local/lib/python` and the **FFC** executable in `~/local/bin`. If you use this option, make sure to set the environment variable `PYTHONPATH` to `~/local/lib/python` and to add `~/local/bin` to the `PATH` environment variable.

C.1.4 Compiling the demos

To test your installation of **FFC**, enter the subdirectory `src/demo` and compile some of the demonstration forms. With **FFC** installed on your system, just type

```
# ffc Poisson.form
```

to compile the bilinear and linear forms for Poisson's equation. This will generate a C++ header file called `Poisson.h` containing UFC [2, 3] code that can be used to assemble the linear system for Poisson's equation.

It is also possible to compile the forms in `src/demo` without needing to install **FFC** on your system. In that case, you need to supply the path to the **FFC** executable:

```
# ../bin/ffc Poisson.form
```

C.1.5 Verifying the generated code

To verify the output generated by the compiler, enter the sub directory `src/test/regression` from within the **FFC** source tree and run the script `test.py`

```
# python test.py
```

This script compiles all forms found in `src/demo` and compares the output with previously compiled forms in `src/test/regression/reference`.

C.2 Debian (Ubuntu) package

In preparation.

Appendix D

Contributing code

If you have created a new module, fixed a bug somewhere, or have made a small change which you want to contribute to **FFC**, then the best way to do so is to send us your contribution in the form of a patch. A patch is a file which describes how to transform a file or directory structure into another. The patch is built by comparing a version which both parties have against the modified version which only you have. Patches can be created with Mercurial or `diff`.

D.1 Creating bundles/patches

D.1.1 Creating a Mercurial (hg) bundle

Creating bundles is the preferred way of submitting patches. It has several advantages over plain diffs. If you are a frequent contributor, consider publishing your source tree so that the **FFC** maintainers (and other users) may pull your changes directly from your tree.

A bundle contains your contribution to **FFC** in the form of a binary patch file generated by Mercurial [1], the revision control system used by **FFC**. Follow the procedure described below to create your bundle.

1. Clone the **FFC** repository:

```
# hg clone http://www.fenics.org/hg/ffc
```

2. If your contribution consists of new files, add them to the correct location in the **FFC** directory tree. Enter the **FFC** directory and add these files to the local repository by typing:

```
# hg add <files>
```

   where `<files>` is the list of new files. You do not have to take any action for previously existing files which have been modified. Do not add temporary or binary files.

3. Enter the **FFC** directory and commit your contribution:

```
# hg commit -m "<description>"
```

   where `<description>` is a short description of what your patch accomplishes.

4. Create the bundle:

```
# hg bundle ffc-<identifier>-<date>.hg
  http://www.fenics.org/hg/ffc
```

   written as one line, where `<identifier>` is a keyword that can be used to identify the bundle as coming from you (your username, last name, first name, a nickname etc) and `<date>` is today's date in the format `yyyy-mm-dd`.
   The bundle now exists as `ffc-<identifier>-<date>.hg`.

When you add your contribution at point 2, make sure that only the files that you want to share are present by typing:

```
# hg status
```

This will produce a list of files. Those marked with a question mark are not tracked by Mercurial. You can track them by using the `add` command as shown above. Once you have added these files, their status changes form `?` to `A`.

D.1.2 Creating a standard (diff) patch file

The tool used to create a patch is called `diff` and the tool used to apply the patch is called `patch`.

Here's an example of how it works. Start from the latest release of **FFC**, which we here assume is release x.y.z. You then have a directory structure under `ffc-x.y.z` where you have made modifications to some files which you think could be useful to other users.

1. Clean up your modified directory structure to remove temporary and binary files which will be rebuilt anyway:

```
# make clean
```

2. From the parent directory, rename the **FFC** directory to something else:

```
# mv ffc-x.y.z ffc-x.y.z-mod
```

3. Unpack the version of **FFC** that you started from:

```
# tar zxfv ffc-x.y.z.tar.gz
```

4. You should now have two **FFC** directory structures in your current directory:

```
# ls
ffc-x.y.z
ffc-x.y.z-mod
```

5. Now use the `diff` tool to create the patch:

```
# diff -u --new-file --recursive ffc-x.y.z
  ffc-x.y.z-mod > ffc-<identifier>-<date>.patch
```

written as one line, where `<identifier>` is a keyword that can be used to identify the patch as coming from you (your username, last name, first name, a nickname etc) and `<date>` is today's date in the format `yyyy-mm-dd`.

6. The patch now exists as `ffc-<identifier>-<date>.patch` and can be distributed to other people who already have `ffc-x.y.z` to easily create your modified version. If the patch is large, compressing it with for example `gzip` is advisable:

```
# gzip ffc-<identifier>-<date>.patch
```

D.2 Sending bundles/patches

Patch and bundle files should be sent to the **FFC** mailing list at the address

```
ffc-dev@fenics.org
```

Include a short description of what your patch/bundle accomplishes. Small patches/bundles have a better chance of being accepted, so if you are making a major contribution, please consider breaking your changes up into several small self-contained patches/bundles if possible.

D.3 Applying changes

D.3.1 Applying a Mercurial bundle

You have received a patch in the form of a Mercurial bundle. The following procedure shows how to apply the patch to your version of **FFC**.

1. Before applying the patch, you can check its content by entering the **FFC** directory and typing:

   ```
   # hg incoming -p
     bundle://<path>/ffc-<identifier>-<date>.hg
   ```

   written as one line, where `<path>` is the path to the bundle. `<path>` can be omitted if the bundle is in the **FFC** directory. The option `-p` can be omitted if you are only interested in a short summary of the changesets found in the bundle.

2. To apply the patch to your version of **FFC** type:

   ```
   # hg unbundle <path>/ffc-<identifier>-<date>.hg
   ```

   followed by:

   ```
     # hg update
   ```

D.3.2 Applying a standard patch file

Let's say that a patch has been built relative to **FFC** release x.y.z. The following description then shows how to apply the patch to a clean version of release x.y.z.

1. Unpack the version of **FFC** which the patch is built relative to:

```
# tar zxfv ffc-x.y.z.tar.gz
```

2. Check that you have the patch `ffc-<identifier>-<date>.patch` and the **FFC** directory structure in the current directory:

```
# ls
ffc-x.y.z
ffc-<identifier>-<date>.patch
```

   Unpack the patch file using `gunzip` if necessary.

3. Enter the **FFC** directory structure:

```
# cd ffc-x.y.z
```

4. Apply the patch:

```
# patch -p1 < ../ffc-<identifier>-<date>.patch
```

   The option `-p1` strips the leading directory from the filename references in the patch, to match the fact that we are applying the patch from inside the directory. Another useful option to `patch` is `--dry-run` which can be used to test the patch without actually applying it.

5. The modified version now exists as `ffc-x.y.z`.

D.4 License agreement

By contributing a patch to **FFC**, you agree to license your contributed code under the GNU General Public License (a condition also built into the GPL license of the code you have modified). Before creating the patch, please update the author and date information of the file(s) you have modified according to the following example:

```
__author__ = "Anders Logg (logg@simula.no)"
__date__ = "2004-11-17 -- 2005-09-09"
__copyright__ = "Copyright (C) 2004, 2005 Anders Logg"
__license__  = "GNU GPL Version 2"

# Modified by Foo Bar 2007
```

As a rule of thumb, the original author of a file holds the copyright.

Appendix E

License

FFC is licensed under the GNU General Public License (GPL) version 2, included verbatim below.

```
    GNU GENERAL PUBLIC LICENSE
       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

    Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
```

```
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
```

language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

```
These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
```

control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you

```
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded.  In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number.  If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation.  If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.
```

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index