FFC User Manual

February 24, 2006

Anders Logg

www.fenics.org

Contents

About this manual

This manual is currently being written. As a consequence, some sections may be incomplete or inaccurate.

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 enumarated from $i = 0$ to $i = n - 1$. Derivatives in $\mathbb{R}^n$ are enumerated similarly: $\partial/\partial x_0, \partial/\partial x_1, \ldots, \partial/\partial x_{n-1}$.

Contact

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

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

Chapter 1

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 B for detailed installation instructions and Chapter 4 for a detailed discussion of the form language.

1.1 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 0.1.0 of **FFC**:

```
# tar zxfv ffc-0.1.0.tar.gz
# cd ffc-0.1.0
# python setup.py install
```

Make sure that you download the latest release (which is not 0.1.0).

Note that you may need to be root on your system to do the last step. You may also need to install the Python packages **FIAT** and Numeric. (See Appendix B for detailed instructions.)

1.2 Compiling Poisson's equation with FFC

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

$$a(v, U) = L(v) \quad \forall v \in \hat{V}_h, \tag{1.1}$$

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

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

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

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

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 (1.1) for a pair of discrete function spaces, specify the forms in a text file with extension `.form`, e.g. `Poisson.form`, as follows:

$\rightarrow$ 已被淘汰了.

```
element = FiniteElement(‘‘Lagrange’’, ‘‘triangle’’, 1)

v = BasisFunction(element)
U = BasisFunction(element)
f = Function(element)

a = dot(grad(v), grad(U))*dx
L = f*v*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 generates the file `Poisson.h` which implements the forms in C++ for inclusion in **DOLFIN**. For help on the `ffc` command, including compilation for other systems than **DOLFIN**, type `ffc -h` or `man ffc`.

Chapter 2

Command-line interface

The command-line interface of **FFC** is documented by the man page for **FFC**, which can be read by the command

```
man ffc
```

on any system where **FFC** has been installed. A copy of this documentation is included below for convenience.

2.1 Synopsis

```
ffc [-h] [-l language] [-d debuglevel] [-f option] input.form
```

2.2 Description

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 representing 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 C++ code for DOLFIN, but this can be changed by specifying a different output language (option -l). It is also possible to add new output languages to FFC.

2.3 Options

2.3.1 `-h, --help`

Display help text and exit.

2.3.2 `-v, --version`

Display version number and exit.

2.3.3 `-l language, --language language`

Specify output language, one of `dolfin` (default), `latex`, `raw`, `ase` or `xml`.

2.3.4 `-d debuglevel, --debug debuglevel`

Specify debug level (default is `0`).

2.3.5 `-f option`

Specify code generation options. The list of options available depends on the specified language (format). Current options include `-f no-gpl` and `-f blas` described in detail below.

2.3.6 `-f no-gpl`

Don't add GPL license to generated code. This option has only effect when compiling with `-ldolfin`.

2.3.7 `-f blas`

Generate code that uses BLAS to compute tensor products. This option has only effect when compiling with `-ldolfin`.

Chapter 3

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 = BasisFunction(element)
U = BasisFunction(element)
f = Function(element)

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

compile([a, L])
```

At the basic level, the only difference between the command-line interface and the Python interface is that the function `compile` must be called when using the Python interface.

In addition to the function `compile`, the Python interface provides the functions `build`, `write` and `writeFiniteElement`. These functions are documented below.

Documentation can also be accessed from within Python. To read the documentation for the function `compile`, run the following commands in a Python shell:

```
from ffc import *
help(compile)
```

3.1 compile(forms, ...)

This function takes as argument a form or list of forms and compiles it into low-level code for assembly. Calling this function is equivalent to first calling `build` followed by `write`.

3.2 build(forms, ...)

This function takes as argument a form or list of forms and does preprocessing of the forms (including computation of the reference tensor), but does not generate any code.

3.3 write(forms, ...)

This function takes a preprocessed form or list of forms and generates code. Note that `build` must be called before `write`:

```
forms = build([a, L])
write(forms)
```

3.4 `writeFiniteElement(element, ...)`

This function generates code for a given `FiniteElement`. Use this function if you just want to generate code for a finite element (including mapping of nodes and nodal points):

```
element = FiniteElement(‘‘Lagrange’’, ‘‘triangle’’, 1)
writeFiniteElement(element)
```

Chapter 4

Form language

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

4.1 Overview

A form is expressed using a combination of basic data types and operators. **FFC** compiles a given multilinear form

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \rightarrow \mathbb{R} \tag{4.1}$$

into code that can be used to compute the corresponding tensor

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r). \tag{4.2}$$

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 functions 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

$$a(v, U) = \int_\Omega v \, \frac{\partial U}{\partial x_0} \, \mathrm{d}x. \tag{4.3}$$

Note 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 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*f*dx` defines a *linear form*, namely $L(v) = \int_\Omega v \, f \, \mathrm{d}x$.

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*.

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.

4.2 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 4.9 below) are allowed in the definition of a form.

4.3 Basic data types

4.3.1 `FiniteElement`

The data type `FiniteElement` represents a finite element on a triangle or tetrahedron. A `FiniteElement` is declared by specifying the type of element, the underlying shape, the polynomial order and, optionally, the number of vector components:

```
element = FiniteElement(type, shape, <degree>, <num_components>)
```

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

- `‘‘Lagrange’’`, representing a standard Lagrange finite element for continuous piecewise polynomial functions;
- `‘‘Discontinuous Lagrange’’`, representing a discontinuous Lagrange finite element for discontinuous piecewise polynomial functions;
- `‘‘Vector Lagrange’’`, representing a standard vector Lagrange finite element for continuous piecewise polynomial vector-valued functions;
- `‘‘Discontinuous vector Lagrange’’`, representing a discontinuous Lagrange finite element for discontinuous piecewise polynomial vector-valued functions.

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

The argument `num_components` is optional and specifies the number of vector components for a vector-valued element. If not specified, the number of vector components is assumed to be the same the dimension d of the underlying shape.

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)
```

4.3.2 `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 sequence of `FiniteElement`s. The following example illustrates how to create a Taylor-Hood element (quadratic velocity and linear pressure):

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

4.3.3 `BasisFunction`

The data type `BasisFunction` represents a basis function on a given finite element. A `BasisFunction` must be declared from a previously declared `FiniteElement`:

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

Note that more than one `BasisFunction` can be declared from the same `FiniteElement`.

Note that the order in which `BasisFunction`s are declared is important. The order determines the order of arguments to the multilinear form. Thus, for a bilinear form $a(v, U)$, the test function v should be declared before the trial function U:

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

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

```
(v, q) = BasisFunctions(TH)
(U, P) = BasisFunctions(TH)
```

4.3.4 `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 from a previously declared `FiniteElement`:

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

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

```
v = BasisFunction(element)
U = BasisFunction(element)
f = 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.

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

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

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

4.3.5 `Constant`

The data type `Constant` represents a constant scalar value that is unknown at compile-time. A `Constant` is declared without any arguments:

```
c = Constant()
```

Just as with `Functions`, it is the responsibility of the user or the system for which the form is compiled to supply the value of the constant at run-time. In the case of **DOLFIN**, a constant is automatically assigned a value from a given user-defined variable.

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

4.3.6 `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)
```

When using the command-line interface to **FFC**, a sequence of free indices are automatically declared 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.

4.3.7 `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)
```

4.4 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 see the comment below).

4.4.1 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.

4.4.2 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.

4.4.3 Scalar multiplication: `*`

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

4.4.4 Scalar division: `/`

Division is not allowed 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. Division by `Function`s and `Constant`s may be implemented in future versions of **FFC**.

4.5 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.

4.5.1 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`.

4.5.2 Scalar product: `dot(v, w)`

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

$$\texttt{dot(v, w)} \leftrightarrow v \cdot w = \sum_{i=0}^{n-1} v_i w_i. \tag{4.4}$$

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

4.5.3 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:

$$\texttt{cross(v, w)} \leftrightarrow v \times w = (v_1 w_2 - v_2 w_1, v_2 w_0 - v_0 w_2, v_0 w_1 - v_1 w_0). \tag{4.5}$$

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

4.5.4 Matrix product: `mult(A, B)`

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

4.5.5 Transpose: `transp(A)`

The operator `transp` accept as argument a matrix and return the transpose of the given matrix:

$$\texttt{transp(A)[i][j]} \leftrightarrow (A^\top)_{ij} = A_{ji}. \tag{4.6}$$

4.5.6 Trace: `trace(A)`

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

$$\texttt{trace}(\mathrm{A}) \leftrightarrow \mathrm{trace}(A) = \sum_{i=0}^{n-1} A_{ii}. \tag{4.7}$$

4.5.7 Vector length: `len(v)`

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

4.5.8 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.

4.5.9 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 `lists`. 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 Numeric `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.

4.6 Differential operators

4.6.1 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:

$$\texttt{D(v, i)} \leftrightarrow \frac{\partial v}{\partial x_i}. \tag{4.8}$$

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.

4.6.2 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:

$$\texttt{grad(v)} \leftrightarrow \mathrm{grad(v)} = \nabla v = (\frac{\partial v}{\partial x_0}, \frac{\partial v}{\partial x_1}, \ldots, \frac{\partial v}{\partial x_{d-1}}). \tag{4.9}$$

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

$$\texttt{grad(v)}[i][j] \leftrightarrow (\mathrm{grad(v)})_{ij} = (\nabla v)_{ij} = \frac{\partial v_i}{\partial x_j}. \tag{4.10}$$

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`.

4.6.3 Divergence: `div(v)`

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

$$\texttt{div(v)} \leftrightarrow \mathrm{div}(v) = \nabla \cdot v = \sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i}. \tag{4.11}$$

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`.

4.6.4 Rotation: `rot(v)`

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

$$\texttt{rot(v)} \leftrightarrow \mathrm{rot}(v) = \nabla \times v = (\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}). \tag{4.12}$$

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

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

4.7 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 or the boundary $\partial\Omega$ of Ω.

4.7.1 Integration over the interior: `*dx`

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

```
dx = Integral(``interior'')
```

If `v` is a scalar-valued expression, then the integral of `v` over the interior of Ω is written as `v*dx`.

When using the command-line interface to **FFC**, the measure `dx` is automatically declared as an integral over the interior of Ω. Note however that a user is free to declare measures with other names or even reuse the variable `dx` for something else.

4.7.2 Integration over the boundary: `*ds`

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

```
ds = Integral(``boundary'')
```

If `v` is a scalar-valued expression, then the integral of `v` over the boundary of Ω is written as `v*ds`.

When using the command-line interface to **FFC**, the measure `ds` is automatically declared as an integral over the boundary of Ω. Note however that a user is free to declare measures with other names or even reuse the variable `ds` for something else.

At this point, complete support has not been added to **FFC** for boundary integrals, which means that all boundary integrals are currently evaluated to zero.

4.8 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`:

$$\texttt{v[i]*w[i]} \leftrightarrow \sum_{i=0}^{n-1} v_i w_i, \tag{4.13}$$

$$\texttt{D(v, i)*D(w, i)} \leftrightarrow \sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i}\frac{\partial w}{\partial x_i} = \nabla v \cdot \nabla w, \tag{4.14}$$

$$\texttt{D(v[i], i)} \leftrightarrow \sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i} = \nabla \cdot v, \tag{4.15}$$

$$\texttt{D(v[i], j)*D(w[i], j)} \leftrightarrow \sum_{i=0}^{n-1}\sum_{j=0}^{d-1} \frac{\partial v_i}{\partial x_j}\frac{\partial w_i}{\partial x_j}. \tag{4.16}$$

Index notation is used internally by **FFC** to represent multilinear forms and in most cases, **FFC** is capable of generating an efficient tensor representation of any given expression. However, in some cases index notation might generate more efficient code.

4.9 User-defined operators

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

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top). \tag{4.17}$$

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 5

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`, which is always the case if the command-line interface is used.

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

5.1 The mass matrix

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

$$a(v, U) = \int_\Omega v\, U \,\mathrm{d}x, \tag{5.1}$$

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

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

v = BasisFunction(element)
```

```
U = BasisFunction(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.

5.2 Poisson's equation

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

$$a(v, U) = \int_\Omega \nabla v \cdot \nabla U \,\mathrm{d}x, \quad (5.2)$$

$$L(v) = \int_\Omega v\, f \,\mathrm{d}x, \quad (5.3)$$

can be implemented as follows:

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

v = BasisFunction(element)
U = BasisFunction(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.

5.3 Vector-valued Poisson

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

$$a(v, U) = \int_\Omega \nabla v : \nabla U \,\mathrm{d}x, \tag{5.4}$$

$$L(v) = \int_\Omega v \cdot f \,\mathrm{d}x, \tag{5.5}$$

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

```
element = FiniteElement(‘‘Vector Lagrange’’, ‘‘triangle’’, 1)

v = BasisFunction(element)
U = BasisFunction(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.

5.4 The strain-strain term of linear elasticity

The strain-strain term of linear elasticity,

$$a(v, U) = \int_\Omega \epsilon(v) : \epsilon(U) \,\mathrm{d}x, \tag{5.6}$$

where

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top) \tag{5.7}$$

can be implemented as follows:

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

v = BasisFunction(element)
U = BasisFunction(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.

5.5 The nonlinear term of Navier–Stokes

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

$$a(v, U) = \int_\Omega v \cdot ((w \cdot \nabla)U)\,\mathrm{d}x, \tag{5.8}$$

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 = BasisFunction(element)
U = BasisFunction(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.

5.6 The heat equation

Discretizing the heat equation,

$$\dot{u} - \nabla \cdot (c\nabla u) = f, \tag{5.9}$$

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

$$\frac{1}{k_n}\int_\Omega (U^n - U^{n-1})\, v \,\mathrm{d}x + \int_\Omega c\,\nabla U^n \cdot \nabla v \,\mathrm{d}x = \int_\Omega f^n\, v\,\mathrm{d}x \tag{5.10}$$

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 $a(v,U) = L(v)$ for all v, we obtain the following pair of bilinear and linear forms:

$$\begin{aligned} a(v, U^n) &= \int_\Omega v\, U^n \,\mathrm{d}x + k_n \int_\Omega c\, v \cdot \nabla U^n \,\mathrm{d}x, && (5.11)\\ L(v) &= \int_\Omega v\, U^{n-1} \,\mathrm{d}x + k_n \int_\Omega v\, f^n \,\mathrm{d}x, && (5.12)\end{aligned}$$

which can be implemented as follows:

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

v  = BasisFunction(element) # Test function
U1 = BasisFunction(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
```

5.7 Mixed formulation of Stokes

To solve Stokes' equations,

$$\begin{aligned} -\Delta u + \nabla p &= f, && (5.13)\\ \nabla \cdot u &= 0, && (5.14) \end{aligned}$$

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:

$$\begin{aligned} a(v,(U,P)) &= \int_\Omega \nabla v : \nabla U - (\nabla \cdot v)\, P + q\,(\nabla \cdot U)\,\mathrm{d}x, && (5.15)\\ L(v) &= \int_\Omega v \cdot f\,\mathrm{d}x. && (5.16) \end{aligned}$$

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) = BasisFunctions(TH)
```

```
(U, P) = BasisFunctions(TH)

f = Function(P2)

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

Appendix A

Reference elements

A.1 The reference triangle

The reference triangle (Figure A.1) is defined by the following three vertices:

$$\begin{aligned} v^0 &= (0, 0), \\ v^1 &= (1, 0), \\ v^2 &= (0, 1). \end{aligned} \tag{A.1}$$

Note that this corresponds to a counter-clockwise orientation of the vertices in the plane.

The edges of the reference triangle are ordered following the convention that edge e^i should be opposite to vertex v^i for $i = 0, 1, 2$, with the vertices of each edge ordered to give a counter-clockwise orientation of the triangle in the plane:

$$\begin{aligned} e^0 &: (v^1, v^2), \\ e^1 &: (v^2, v^0), \\ e^2 &: (v^0, v^1). \end{aligned} \tag{A.2}$$

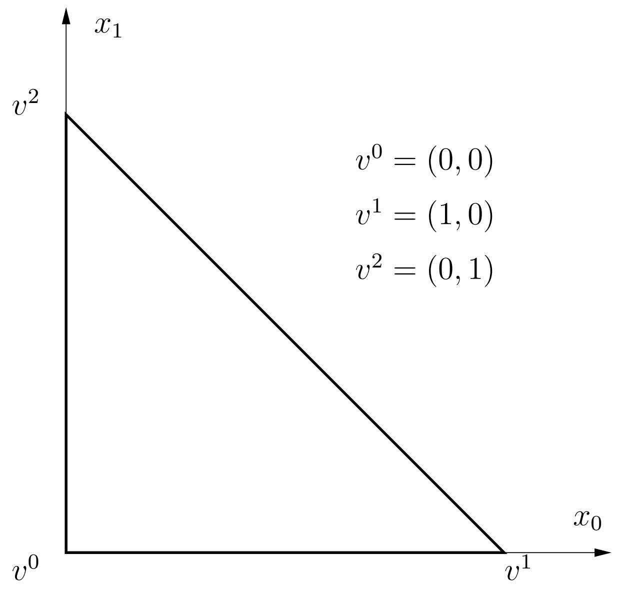

Figure A.1: Physical coordinates of the reference triangle.

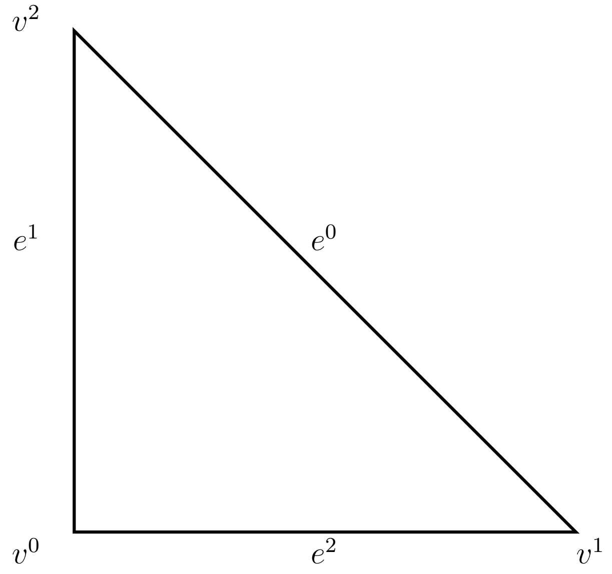

Figure A.2: Ordering of mesh entities (vertices and edges) for the reference triangle.

A.2 The reference tetrahedron

The reference tetrahedron (Figure A.3) is defined by the following four vertices:

$$\begin{aligned} v^0 &= (0,0,0), \\ v^1 &= (1,0,0), \\ v^2 &= (0,1,0), \\ v^4 &= (0,0,1). \end{aligned} \tag{A.3}$$

The faces of the reference tetrahedron are ordered following the convention that face f^i should be opposite to vertex v^i for $i = 0, 1, 2, 3$, with the vertices of each face ordered to give a counter-clockwise orientation of each face as seen from the outside of the tetrahedron and the first vertex of face f^i given by vertex $v^{i+1 \mod 4}$:

$$\begin{aligned} f^0 &: (v^1, v^3, v^2), \\ f^1 &: (v^2, v^3, v^0), \\ f^2 &: (v^3, v^1, v^0), \\ f^3 &: (v^0, v^1, v^2). \end{aligned} \tag{A.4}$$

The edges of the reference tetrahedron are ordered following the convention that edges e^0, e^1, e^2 should correspond to the edges of the reference triangle. Edges e^3, e^4, e^5 all ending up at vertex v^3 are ordered based on their first vertex:

$$\begin{aligned} e^0 &: (v^1, v^2), \\ e^1 &: (v^2, v^0), \\ e^2 &: (v^0, v^1), \\ e^3 &: (v^0, v^3), \\ e^4 &: (v^1, v^3), \\ e^5 &: (v^2, v^3). \end{aligned} \tag{A.5}$$

The ordering of vertices on faces implicitly defines an ordering of edges on faces by identifying an edge on a face with the opposite vertex on the face:

$$\begin{aligned} f^0 &: (e^5, e^0, e^4), \\ f^1 &: (e^3, e^1, e^5), \\ f^2 &: (e^2, e^3, e^4), \\ f^3 &: (e^0, e^1, e^2). \end{aligned} \tag{A.6}$$

Note that the ordering of edges on f^3 is the same as the ordering of edges on the reference triangle. Also note that the internal ordering of vertices on edges does not always follow the orientation of the face (which is not possible).

A.3 Ordering of degrees of freedom

The local and global orderings of degrees of freedom or *nodes* are obtained by associating each node with a mesh entity, locally and globally.

A.3.1 Mesh entities

We distinguish between mesh entities of different topological dimensions:

vertices	topological dimension 0
edges	topological dimension 1
faces	topological dimension 2
cells	topological dimension 2 or 3

A cell can be either a triangle or a tetrahedron depending on the type of mesh. For a mesh consisting of triangles, the mesh entities involved are vertices, edges and cells, and for a mesh consisting of tetrahedrons, the mesh entities involved are vertices, edges, faces and cells.

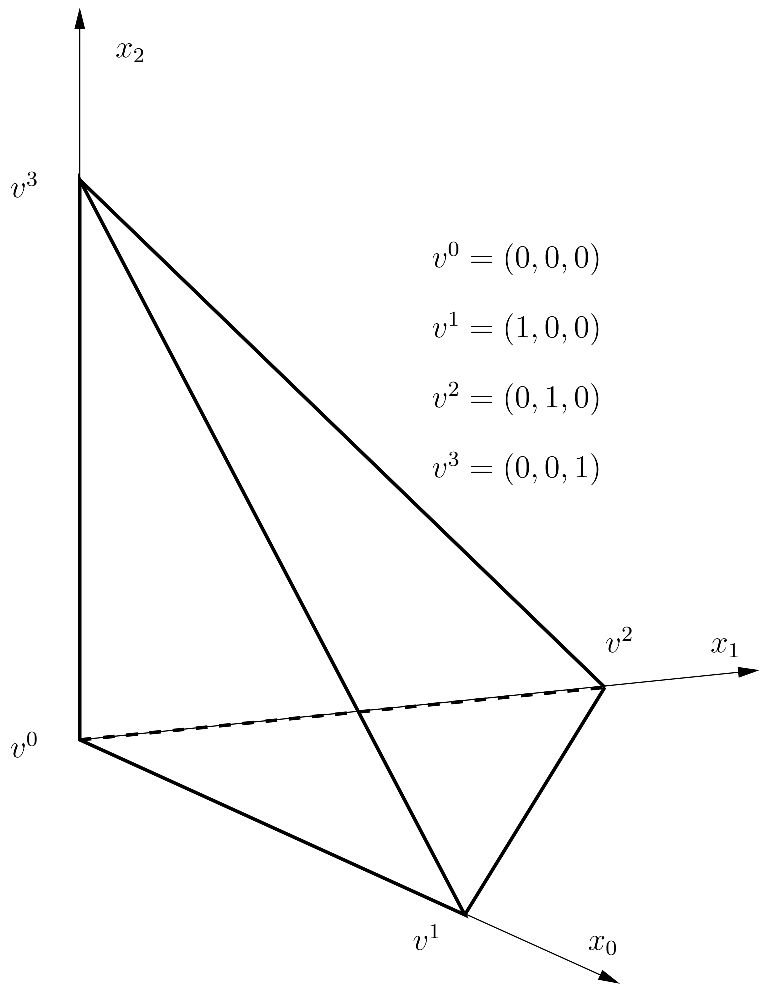

Figure A.3: Physical coordinates of the reference tetrahedron.

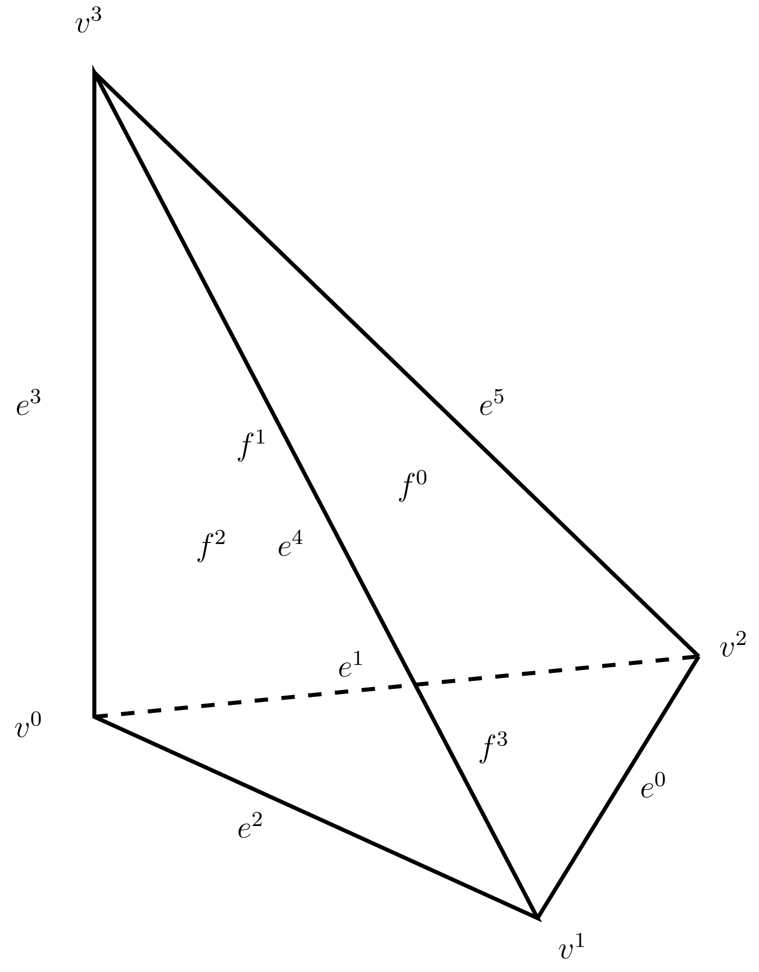

Figure A.4: Ordering of mesh entities (vertices, edges, faces) for the reference tetrahedron.

A.3.2 Ordering among mesh entities

With each mesh entity, there can be associated zero or more nodes and the nodes are ordered locally and globally based on the topological dimension of the mesh entity with which they are associated. Thus, any nodes associated with vertices are ordered first and nodes associated with cells last.

If more than one node is associated with a single mesh entity, the internal ordering of the nodes associated with the mesh entity becomes important, in particular for edges and faces, where the nodes of two adjacent cells sharing a common edge or face must lign up.

A.3.3 Internal ordering on edges

For edges containing more than one node, the nodes are ordered in the direction from the first vertex (v_e^0) of the edge to the second vertex (v_e^1) of the edge as in Figure A.5.

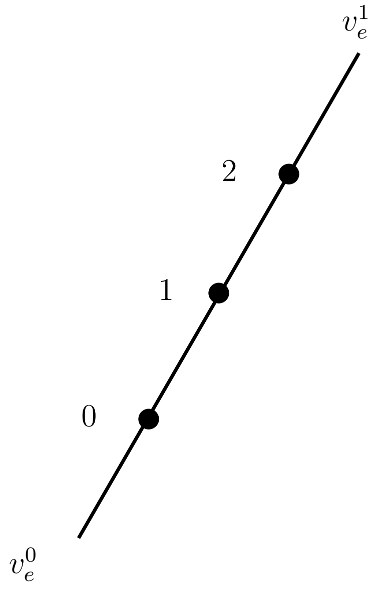

Figure A.5: Internal ordering of nodes on edges.

A.3.4 Alignment of edges

Depending on the orientation of any given cell, an edge on the cell may be aligned or not aligned with the corresponding edge on the reference cell if the vertices of the cell are mapped to the reference cell. We define the *alignment* of an edge with respect to a cell to be 0 if the edge is aligned with the orientation of the reference cell and 1 otherwise.

Example 1: The alignment of the first edge (e^0) on a triangle is 0 if the first vertex of the edge is the second vertex (v^1) of the triangle.

Example 2: The alignment of the second edge (e^1) on a tetrahedron is 0 if the first vertex of the edge is the third vertex (v^2) of the tetrahedron.

If two cells share a common edge and the edge is aligned with one of the cells and not the other, we must reverse the order in which the local nodes are mapped to global nodes on one of the two cells. As a convention, the order is kept if the alignment is 0 and reversed if the alignment is 1.

A.3.5 Internal ordering on faces

For faces containing more than one node, the ordering of nodes is nested going from the first to the third vertex and in each step going from the first to the second vertex as in Figure A.6.

A.3.6 Alignment of faces

There are six different ways for a face to be aligned on a tetrahedron; there are three ways to pick the first edge of the face, and once the first edge is picked, there are two ways to pick the second edge. To define an alignment of faces as an integer between 0 and 5, we compare the ordering of edges on a face with the ordering of edges on the corresponding face on the reference tetrahedron. If the first edge of the face matches the first edge on the corresponding face on the reference tetrahedron and also the second edge matches the second edge on the reference tetrahedron, then the alignment is 0. If only the first

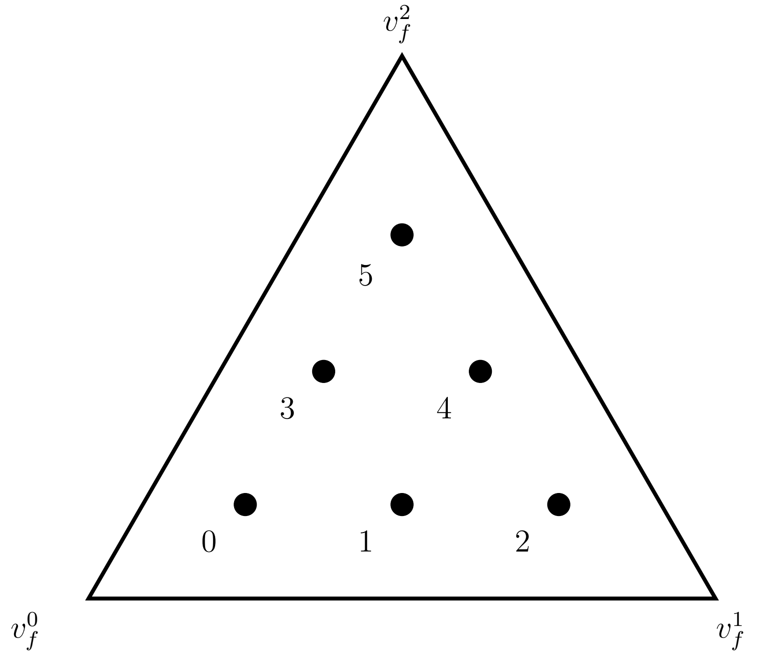

Figure A.6: Internal ordering of nodes on faces.

edge matches, then the alignment is 1. We similarly define alignments $2, 3$ by matching the first and second edges with the second and third edges on the corresponding face on the reference tetrahedron, and alignments $4, 5$ by matching the first and second edges with the third and first edges on the corresponding face on the reference tetrahedron.

Example 1: The alignment of the first face of a tetrahedron is 0 if the first edge of the face is edge number 5 and the second edge is edge number 0.

Example 2: The alignment of the first face of a tetrahedron is 1 if the first edge of the face is edge number 5 and the second edge is not edge number 0. (It must then be edge number 4.)

Example 3: The alignment of the first face of a tetrahedron is 4 if the first edge of the face is edge number 4 and the second edge is edge number 5.

Example 4: The alignment of the first face of a tetrahedron is 5 if the first edge of the face is edge number 4 and the second edge is not edge number 5. (It must then be edge number 0.)

Appendix B

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 effort is underway to provide precompiled Debian packages of **FFC** and other **FEniCS** components.

B.1 Installing from source

B.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 Numeric module. In addition, you need to have a working Python installation on your system.

Installing Python

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

```
# python -V
Python 2.4.1
```

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 users, the package to install is `python2.4`.

Installing Numeric

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

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

For Debian users, the package to install is `python2.4-numeric`.

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.

In addition, you will need to install the Python package LinearAlgebra, which may already be included in your installation of Python Numeric. For Debian users, the package to install is `python2.4-numeric-ext`.

B.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-0.1.0.tar.gz`, and unpack using the command

```
# tar zxfv ffc-0.1.0.tar.gz
```

This creates a directory `ffc-0.1.0` containing the **FFC** source code.

If you want the very latest version of **FFC**, there is also a version named `ffc-cvs-current.tar.gz` which provides a snapshot of the current CVS version of **FFC**, updated automatically from the CVS repository each hour. This version may contain features not yet present in the latest release, but may also be less stable and even not work at all.

B.1.3 Installing the compiler

FFC follows the standard 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 in the directory `/usr/lib/python2.4/site-packages`). In addition, the compiler executable (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.

B.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` that can be used with **DOLFIN** to implement a solver for Poisson's equation. Adding the flag `-l latex` generates output in LaTeX format:

```
# ffc -l latex Poisson.form
# latex Poisson.tex
# xdvi Poisson.dvi
```

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
```

B.1.5 Verifying the generated code

To verify the output generated by the compiler, run the script `verify` from within the **FFC** source tree:

```
# scripts/verify
```

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

B.2 Debian package

In preparation.

Appendix C

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.

C.1 Creating a patch

The tool used to create a patch is called `diff` and the tool used to apply the patch is called `patch`. These tools are free software and are standard on most Unix systems.

Here's an example of how it works. Start from the latest release of **FFC**, which we here assume is release 0.1.0. You then have a directory structure under `ffc-0.1.0` 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-0.1.0 ffc-0.1.0-mod
```

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

```
# tar zxfv ffc-0.1.0.tar.gz
```

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

```
# ls
ffc-0.1.0
ffc-0.1.0-mod
```

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

```
# diff -u --new-file --recursive ffc-0.1.0
ffc-0.1.0-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-0.1.0` to easily create your modified version. If the patch is large, compressing it with for example `gzip` is advisable:

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

C.2 Sending patches

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

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

Include a short description of what your patch accomplishes. Small patches 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 if possible.

C.3 Applying a patch (maintainers)

Let's say that a patch has been built relative to **FFC** release 0.1.0. The following description then shows how to apply the patch to a clean version of release 0.1.0.

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

   ```
   # tar zxfv ffc-0.1.0.tar.gz
   ```

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

   ```
   # ls
   ffc-0.1.0
   ffc-<identifier>-<date>.patch
   ```

   Unpack the patch file using `gunzip` if necessary.

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

   ```
   # cd ffc-0.1.0
   ```

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-0.1.0`.

C.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@tti-c.org)’’
__date__ = ‘‘2004-11-17 -- 2005-09-09’’
__copyright__ = ‘‘Copyright (c) 2004, 2005 Anders Logg’’
__license__  = ‘‘GNU GPL Version 2’’
# Modified by Johan Jansson 2005.
```

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

Appendix D

License

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

Index