UFL Specification and User Manual 0.3

November 16, 2010

Martin S. Alnæs, Anders Logg

www.fenics.org

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

`ufl-dev@fenics.org`

Chapter 1

Introduction

The Unified Form Language (**UFL**) is a domain specific language for defining discrete variational forms and functionals in a notation close to pen-and-paper formulation.

UFL [2] is part of the **FEniCS** project [4], and is usually used in combination with other components from this project to compute solutions to partial differential equations. The form compilers **FFC** [6] and **SFC** [1] use **UFL** as their end-user interface, producing implementations of the **UFC** [3] interface as their output. See the **DOLFIN** manual [5] for more details about using **UFL** in an integrated problem solving environment.

This manual is intended for different audiences. If you are an end user and all you want to do is to solve your PDEs with the **FEniCS** framework, Chapters 2 and 3 are for you. These two chapters explain how to use all operators available in the language and present a number of examples to illustrate the use of the form language in applications. The rest of the chapters contain more technical details intended for developers who need to understand what is happening behind the scenes and modify or extend **UFL** in the future.

Chapter 4 details the implementation of the language, in particular how expressions are represented internally by **UFL**. This can also be useful knowledge to understand error messages and debug errors in your form files.

Chapter 5 explains many algorithms to work with **UFL** expressions, mostly intended to aid developers of form compilers. The algorithms available includes helper functions for easy and efficient iteration over expression trees, formatting tools to present expressions as text or images of different kinds, utilities to analyse properties of expressions or checking their validity, automatic differentiation algorithms, as well as algorithms to work with the computational graphs of expressions.

Chapter 2

Form Language

UFL consists of a set of operators and atomic expressions that can be used to express variational forms and functionals. Below we will define all these operators and atomic expressions in detail.

UFL is built on top of, or embedded in, the high level language Python. 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 2.12 below) are useful in the definition of a form. However, it is usually a good idea to avoid using advanced Python features in the form definition, to stay close to the mathematical notation.

The entire form language can be imported in Python with the line

```
from ufl import *
```

which is assumed in all examples below and can be omitted in `.ufl` files. This can be useful for experimenting with the language in an interactive Python interpreter.

2.1 Forms and Integrals

UFL is designed to express forms in the following generalized format:

$$\begin{aligned} a(v_1, \ldots, v_r; w_1, \ldots, w_n) = & \\ & \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c(v_1, \ldots, v_r; w_1, \ldots w_n) \, \mathrm{d}x \\ & + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}s \\ & + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}S. \end{aligned} \tag{2.1}$$

Here the form a depends on the *form arguments* $v_1, \ldots, v_r$ and the *form coefficients* $w_1, \ldots, w_n$, and its expression is a sum of integrals. Each term of a valid form expression must be a scalar-valued expression integrated exactly once. How to define form arguments and integrand expressions is detailed in the rest of this chapter.

Integrals are expressed through multiplication with a measure, representing an integral over either of

- the interior of the domain Ω ($\mathrm{d}x$, cell integral);
- the boundary $\partial\Omega$ of Ω ($\mathrm{d}s$, exterior facet integral);
- the set of interior facets Γ ($\mathrm{d}S$, interior facet integral).

UFL declares the measures `dx` $\leftrightarrow$ $\mathrm{d}x$, `ds` $\leftrightarrow$ $\mathrm{d}s$, and `dS` $\leftrightarrow$ $\mathrm{d}S$.

As a basic example, assume `v` is a scalar-valued expression and consider the integral of `v` over the interior of Ω. This may be expressed as

```
a = v*dx
```

and the integral of `v` over $\partial\Omega$ is written as

```
a = v*ds
```

Alternatively, measures can be redefined to represent numbered subsets of a domain, such that a form can take on different expressions on different parts of the domain. If `c`, `e0` and `e1` are scalar-valued expressions, then

```
a = c*dx + e0*ds(0) + e1*ds(1)
```

represents

$$a = \int_\Omega c\,dx + \int_{\partial\Omega_0} e_0\,ds + \int_{\partial\Omega_1} e_1\,ds.$$

where

$$\partial\Omega_0 \subset \partial\Omega, \qquad \partial\Omega_1 \subset \partial\Omega.$$

Generalizing this further we end up with the expression (2.1). Note that the domain Ω and its subdomains and boundaries are not known to **UFL**. These will not enter the stage until you start using **UFL** in a problem solving environment like **DOLFIN**.

[**Advanced**] A feature for advanced users is attaching metadata to integrals. This can be used to define different quadrature degrees for different terms in a form, and to override other form compiler specific options separately for different terms.

```
a = c0*dx(0, metadata0) + c1*dx(1, metadata1)
```

The convention is that metadata should be a dict with any of the following keys:

- `"integration_order"`: Integer defining the polynomial order that should be integrated exactly. This is a compilation hint, and the form compiler is free to ignore this if for example exact integration is being used.
- `"ffc"`: A dict with further FFC specific options, see the FFC manual.

- `"sfc"`: A dict with further SFC specific options, see the SFC manual.
- Other string: A dict with further options specific to some other external code.

Other standardized options may be added in later versions.

```
metadata0 = {"ffc": {"representation": "quadrature"}}
metadata1 = {"integration_order": 7,
             "ffc": {"representation": "tensor"}}

a = v*u*dx(0, metadata1) + f*v*dx(0, metadata2)
```

2.2 Finite Element Spaces

Before we can explain how form arguments are declared, we need to show how to define function spaces. **UFL** can represent very flexible general hierarchies of mixed finite elements, and has predefined names for most common element families.

2.2.1 Cells

A polygonal cell is defined by a basic shape and a degree[1], written like

```
cell = Cell(shape, degree)
```

Valid shapes are ”interval”, ”triangle”, ”tetrahedron”, ”quadrilateral”, and ”hexahedron”. Some examples:

[1]Note that the other components of FEniCS does not yet handle cells of higher degree, so this will only be useful in the future.

```
# Cubic triangle cell
cell = Cell("triangle", 3)

# Quadratic tetrahedron cell
cell = Cell("tetrahedron", 2)
```

Objects for linear cells of all basic shapes are predefined:

```
# Predefined linear cells
cell = interval
cell = triangle
cell = tetrahedron
cell = quadrilateral
cell = hexahedron
```

In the rest of this document, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible. Using a variable `cell` to hold the cell type used in a form is highly recommended, since this makes most form definitions dimension independent.

2.2.2 Element Families

UFL predefines a set of names of known element families. When defining a finite element below, the argument `family` is a string and its possible values include:

- `"Lagrange"` or `"CG"`, representing standard scalar Lagrange finite elements (continuous piecewise polynomial functions);
- `"Discontinuous Lagrange"` or `"DG"`, 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 $H(\text{div})$ elements;

- `"Brezzi-Douglas-Fortin-Marini"` or `"BDFM"`, representing vector-valued Brezzi–Douglas–Fortin–Marini $H(\text{div})$ elements;

- `"Raviart-Thomas"` or `"RT"`, representing vector-valued Raviart–Thomas $H(\text{div})$ elements.

- `"Nedelec 1st kind H(div)"` or `"N1div"`, representing vector-valued Nedelec $H(\text{div})$ elements (of the first kind).

- `"Nedelec 2st kind H(div)"` or `"N2div"`, representing vector-valued Nedelec $H(\text{div})$ elements (of the second kind).

- `"Nedelec 1st kind H(curl)"` or `"N1curl"`, representing vector-valued Nedelec $H(\text{curl})$ elements (of the first kind).

- `"Nedelec 2st kind H(curl)"` or `"N2curl"`, representing vector-valued Nedelec $H(\text{curl})$ elements (of the second kind).

- `"Quadrature"` or `"Q"`, representing artificial "finite elements" with degrees of freedom being function evaluation at quadrature points;

- `"Boundary Quadrature"` or `"BQ"`, representing artificial "finite elements" with degrees of freedom being function evaluation at quadrature points on the boundary;

[**Advanced**] New elements can be added dynamically by the form compiler using the function `register_element`. See the docstring for details. To see which elements are registered (including the standard built in ones listed above) call the function `show_elements`.

2.2.3 Basic Elements

A `FiniteElement`, some times called a basic element, represents a finite element in some family on a given cell with a certain polynomial degree. Valid families and cells are explained above.

The notation is:

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

Some examples:

```
element = FiniteElement("Lagrange", interval, 3)
element = FiniteElement("DG", tetrahedron, 0)
element = FiniteElement("BDM", triangle, 1)
```

2.2.4 Vector Elements

A `VectorElement` represents a combination of basic elements such that each component of a vector is represented by the basic element. The size is usually omitted, the default size equals the geometry dimension.

The notation is:

```
element = VectorElement(family, cell, degree[, size])
```

Some examples:

```
element = VectorElement("CG", triangle, 2)
element = VectorElement("DG", tetrahedron, 0, size=6)
```

2.2.5 Tensor Elements

A `TensorElement` represents a combination of basic elements such that each component of a tensor is represented by the basic element. The shape is usually omitted, the default shape is (d, d) where d is the geometry dimension.

The notation is:

```
element = TensorElement(family, cell, degree[, shape, symmetry])
```

Any shape tuple consisting of positive integers is valid, and the optional symmetry can either be set to `True` which means standard matrix symmetry (like $A_{ij} = A_{ji}$), or a `dict` like { `(0,1):(1,0), (0,2):(2,0)` } where the `dict` keys are index tuples that are represented by the corresponding `dict` value.

Examples:

```
element = TensorElement("CG", cell, 2)
element = TensorElement("DG", cell, 0, shape=(6,6))
element = TensorElement("DG", cell, 0, symmetry=True)
element = TensorElement("DG", cell, 0, symmetry={(0,0): (1,1)})
```

2.2.6 Mixed Elements

A `MixedElement` represents an arbitrary combination of other elements. `VectorElement` and `TensorElement` are special cases of a `MixedElement` where all subelements are equal.

General notation for an arbitrary number of subelements:

```
element = MixedElement(element1, element2[, element3, ...])
```

Shorthand notation for two subelements:

```
element = element1 * element2
```

NB! Note that multiplication is a binary operator, such that

```
element = element1 * element2 * element3
```

represents `(e1 * e2) * e3`, i.e. this is a mixed element with two subelements `(e1 * e2)` and `e3`.

See section 2.3 for details on how defining functions on mixed spaces can differ from functions on other finite element spaces.

Examples:

```
# Taylor-Hood element
V = VectorElement("Lagrange", cell, 2)
P = FiniteElement("Lagrange", cell, 1)
TH = V * P

# A tensor-vector-scalar element
T = TensorElement("Lagrange", cell, 2, symmetry=True)
V = VectorElement("Lagrange", cell, 1)
P = FiniteElement("DG", cell, 0)
ME = MixedElement(T, V, P)
```

2.2.7 EnrichedElement

The data type `EnrichedElement` represents the vector sum of two (or more) finite elements.

Example: The Mini element can be constructed as

```
P1 = VectorElement("Lagrange", "triangle", 1)
B  = VectorElement("Bubble", "triangle", 3)
Q  = FiniteElement("Lagrange", "triangle", 1)

Mini = (P1 + B) * Q
```

2.3 Form Arguments

Form arguments are divided in two groups, basis functions and functions[2]. A `BasisFunction` represents an arbitrary basis function in a given discrete finite element space, while a `Function` represents a function in a discrete finite element space that will be provided by the user at a later stage. The number of `BasisFunction`s that occur in a `Form` equals the arity of the form.

2.3.1 Basis functions

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:

[2]The term *function* in UFL maps to the term *coefficient* in UFC.

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

For a `BasisFunction` on a `MixedElement` (or `VectorElement` or `TensorElement`), the function `split` can be used to extract basis function values on subspaces, as illustrated here for a mixed Taylor–Hood element:

```
vq = BasisFunction(TH)
v, q = split(up)
```

A shorthand for this is in place called `BasisFunctions`:

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

For convenience, `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). Their usage is otherwise the same as for BasisFunction:

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

2.3.2 Coefficient functions

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 the order in which `Function`s are declared is important, directly reflected in the ordering they have among the arguments to each `Form` they are part of.

`Function` is used to represent user-defined functions, including, e.g., source terms, body forces, variable coefficients and stabilization terms. **UFL** treats each `Function` as a linear combination of unknown basis functions with unknown coefficients, that is, **UFL** knows nothing about the concrete basis functions of the element and nothing about the value of the function.

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)
```

For a `Function` on a `MixedElement` (or `VectorElement` or `TensorElement`), the function `split` can be used to extract function values on subspaces, as illustrated here for a mixed Taylor–Hood element:

```
up = Function(TH)
u, p = split(up)
```

A shorthand for this is in place called `Functions`:

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

Spatially constant (or discontinuous piecewise constant) functions can conveniently be represented by `Constant`, `VectorConstant`, and `TensorConstant`.

```
c0 = Constant(cell)
v0 = VectorConstant(cell)
t0 = TensorConstant(cell)
```

These three lines are equivalent with first defining DG0 elements and then defining a `Function` on each, illustrated here:

```
DG0 = FiniteElement("Discontinuous Lagrange", cell, 0)
DG0v = VectorElement("Discontinuous Lagrange", cell, 0)
DG0t = TensorElement("Discontinuous Lagrange", cell, 0)

c1 = Function(DG0)
v1 = Function(DG0v)
t1 = Function(DG0t)
```

2.4 Basic Datatypes

UFL expressions can depend on some other quantities in addition to the functions and basis functions described above.

2.4.1 Literals and geometric quantities

Some atomic quantities are derived from the cell. For example, the (global) spatial coordinates are available as a vector valued expression `cell.x`:

```
# Linear form for a load vector with a sin(y) coefficient
v = TestFunction(element)
```

```
x = cell.x
L = sin(x[1])*v*dx
```

Another quantity is the (outwards pointing) facet normal `cell.n`. The normal vector is only defined on the boundary, so it can't be used in a cell integral.

Example functional `M`, an integral of the normal component of a function `g` over the boundary:

```
n = cell.n
g = Function(VectorElement("CG", cell, 1))
M = dot(n, g)*ds
```

Python scalars (int, float) can be used anywhere a scalar expression is allowed. Another literal constant type is `Identity` which represents an $n \times n$ unit matrix of given size n, as in this example:

```
# Geometric dimension
d = cell.d

# d x d identiy matrix
I = Identity(d)

# Kronecker delta
delta_ij = I[i,j]
```

[**Advanced**] Note that there are some differences from **FFC**. In particular, using `FacetNormal` or `cell.n` does not implicitly add another coefficient Function to the form, the normal should be automatically computed in **UFC** code. Note also that `MeshSize` has been removed because the meaning is ambiguous (does it mean min, max, avg, cell radius?), so use a `Constant` instead.

2.5 Indexing and tensor components

UFL 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`:

$$\texttt{v[i]*w[i]} \quad \leftrightarrow \quad \sum_{i=0}^{n-1} v_i w_i = \mathbf{v} \cdot \mathbf{w}, \tag{2.2}$$

$$\texttt{Dx(v, i)*Dx(w, i)} \quad \leftrightarrow \quad \sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i} \frac{\partial w}{\partial x_i} = \nabla v \cdot \nabla w, \tag{2.3}$$

$$\texttt{Dx(v[i], i)} \quad \leftrightarrow \quad \sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i} = \nabla \cdot v, \tag{2.4}$$

$$\texttt{Dx(v[i], j)*Dx(w[i], j)} \quad \leftrightarrow \quad \sum_{i=0}^{n-1} \sum_{j=0}^{d-1} \frac{\partial v_i}{\partial x_j} \frac{\partial w_i}{\partial x_j} = \nabla \mathbf{v} : \nabla \mathbf{w}. \tag{2.5}$$

Here we'll try to very briefly summarize the basic concepts of tensor algebra and index notation, just enough to express the operators in **UFL**.

Assuming an Euclidean space in d dimensions with d = 1, 2, or 3, and a set of orthonormal basis vectors $\mathbf{i}_i$ for $i \in 0, \ldots, d-1$, we can define the dot product of any two basis functions as

$$\mathbf{i}_i \cdot \mathbf{i}_j = \delta_{ij}, \tag{2.6}$$

where δ_{ij} is the Kronecker delta

$$\delta_{ij} \equiv \begin{cases} 1, & i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

A rank 1 tensor (vector) quantity $\mathbf{v}$ can be represented in terms of unit vectors and its scalar components in that basis. In tensor algebra it is common to

assume implicit summation over indices repeated twice in a product.

$$\mathbf{v} = v_k \mathbf{i}_k \equiv \sum_k v_k \mathbf{i}_k. \tag{2.8}$$

Similarly, a rank two tensor (matrix) quantity $\mathbf{A}$ can be represented in terms of unit matrices, that is outer products of unit vectors:

$$\mathbf{A} = A_{ij} \mathbf{i}_i \mathbf{i}_j \equiv \sum_i \sum_j A_{ij} \mathbf{i}_i \mathbf{i}_j. \tag{2.9}$$

This generalizes to tensors of arbitrary rank:

$$\mathcal{C} = C_\iota \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}} \tag{2.10}$$

$$\equiv \sum_{\iota_0} \cdots \sum_{\iota_{r-1}} C_\iota \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}}, \tag{2.11}$$

where $\mathcal{C}$ is a rank r tensor and ι is a multiindex of length r.

When writing equations on paper, a mathematician can easily switch between the $\mathbf{v}$ and v_i representations without stating it explicitly. This is possible because of flexible notation and conventions. In a programming language, we can't use the boldface notation which associates $\mathbf{v}$ and v by convention, and we can't always interpret such conventions unambiguously. Therefore, **UFL** requires that an expression is explicitly mapped from its tensor representation ($\mathbf{v}$,$\mathbf{A}$) to its component representation (v_i, A_{ij}) and back. This is done using `Index` objects, the indexing operator (`v[i]`), and the function `as_tensor`. More details on these follow.

In the following descriptions of **UFL** operator syntax, i-l and p-s are assumed to be predefined indices, and unless otherwise specified the name v refers to some vector valued expression, and the name A refers to some matrix valued expression. The name C refers to a tensor expression of arbitrary rank.

2.5.1 Defining indices

A set of indices `i`, `j`, `k`, `l` and `p`, `q`, `r`, `s` are predefined, and these should be enough for many applications. Examples will usually use these objects instead of creating new ones to conserve space.

The data type `Index` represents an index used for subscripting derivatives or taking components of non-scalar expressions. To create indices, you can either make a single using `Index()` or make several at once conveniently using `indices(n)`.

```
i = Index()
j, k, l = indices(3)
```

Each of these represents an *index range* determined by the context; if used to subscript a tensor-valued expression, the range is given by the shape of the expression, 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.

[**Advanced**] If using **UFL** inside PyDOLFIN or another larger programming environment, it is a good idea to define your indices explicitly just before your form uses them, to avoid name collisions. The definition of the predefined indices is simply

```
i, j, k, l = indices(4)
p, q, r, s = indices(4)
```

[**Advanced**] Note that in the old **FFC** notation, the definition

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

meant that the value of the index remained constant. This does not mean the same in **UFL**, and this notation is only meant for internal usage. Fixed indices are simply integers instead:

```
i = 0
```

2.5.2 Taking components of tensors

Basic fixed indexing of a vector valued expression v or matrix valued expression A:

- `v[0]`: component access, representing the scalar value of the first component of v
- `A[0,1]`: component access, representing the scalar value of the first row, second column of A

Basic indexing:

- `v[i]`: component access, representing the scalar value of some component of v
- `A[i,j]`: component access, representing the scalar value of some component i,j of A

More advanced indexing:

- `A[i,0]`: component access, representing the scalar value of some component i of the first column of A
- `A[i,:]`: row access, representing some row i of A, i.e. rank(A[i,:]) == 1
- `A[:,j]`: column access, representing some column j of A, i.e. rank(A[:,j]) == 1
- `C[...,0]`: subtensor access, representing the subtensor of A with the last axis fixed, e.g., A[...,0] == A[:,0]
- `C[j,...]`: subtensor access, representing the subtensor of A with the last axis fixed, e.g., A[j,...] == A[j,:]

2.5.3 Making tensors from components

If you have expressions for scalar components of a tensor and wish to convert them to a tensor, there are two ways to do it. If you have a single expression with free indices that should map to tensor axes, like mapping v_k to $\mathbf{v}$ or A_{ij} to $\mathbf{A}$, the following examples show how this is done.

```
vk = Identity(cell.d)[0,k]
v = as_tensor(vk, (k,))

Aij = v[i]*u[j]
A = as_tensor(Aij, (i,j))
```

Here `v` will represent unit vector $\mathbf{i}_0$, and `A` will represent the outer product of `v` and `u`.

If you have multiple expressions without indices, you can build tensors from them just as easily, as illustrated here:

```
v = as_vector([1.0, 2.0, 3.0])
A = as_matrix([[u[0], 0], [0, u[1]]])
B = as_matrix([[a+b for b in range(2)] for a in range(2)])
```

Here `v`, `A` and `B` will represent the expressions

$$\mathbf{v} = \mathbf{i}_0 + 2\mathbf{i}_1 + 3\mathbf{i}_2, \tag{2.12}$$

$$\mathbf{A} = \begin{bmatrix} u_0 & 0 \\ 0 & u_1 \end{bmatrix}, \tag{2.13}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}. \tag{2.14}$$

Note that the function `as_tensor` generalizes from vectors to tensors of arbitrary rank, while the alternative functions `as_vector` and `as_matrix` work the same way but are only for constructing vectors and matrices. They are included for readability and convenience only.

2.5.4 Implicit summation

Implicit summation can occur in only a few situations. A product of two terms that shares the same free index is implicitly treated as a sum over that free index:

- `v[i]*v[i]`: $\sum_i v_i v_i$
- `A[i,j]*v[i]*v[j]`: $\sum_j (\sum_i A_{ij} v_i) v_j$

A tensor valued expression indexed twice with the same free index is treated as a sum over that free index:

- `A[i,i]`: $\sum_i A_{ii}$
- `C[i,j,j,i]`: $\sum_i \sum_j C_{ijji}$

The spatial derivative, in the direction of a free index, of an expression with the same free index, is treated as a sum over that free index:

- `v[i].dx(i)`: $\sum_i v_i$
- `A[i,j].dx(i)`: $\sum_i \frac{d(A_{ij})}{dx_i}$

Note that these examples are some times written $v_{i,i}$ and $A_{ij,i}$ in pen-and-paper index notation.

2.6 Basic algebraic operators

The basic algebraic operators +, -, *, / can be used freely on **UFL** expressions. They do have some requirements on their operands, summarized here:

Addition or subtraction, `a + b` or `a - b`:

- The operands a and b must have the same shape.
- The operands a and b must have the same set of free indices.

Division, `a / b`:

- The operand b must be a scalar expression.
- The operand b must have no free indices.
- The operand a can be non-scalar with free indices, in which division represents scalar division of all components with the scalar b.

Multiplication, `a * b`:

- The only non-scalar operations allowed is scalar-tensor, matrix-vector and matrix-matrix multiplication.
- If either of the operands have any free indices, both must be scalar.
- If any free indices are repeated, summation is implied.

2.7 Basic nonlinear functions

Some basic nonlinear functions are also available, their meaning mostly obvious.

- `abs(f)`: the absolute value of f.
- `sign(f)`: the sign of f (+1 or -1).
- `pow(f, g)` or `f**g`
- `sqrt(f)`
- `exp(f)`

- `ln(f)`
- `cos(f)`
- `sin(f)`

These functions do not accept non-scalar operands or operands with free indices or `BasisFunction` dependencies.

2.8 Tensor Algebra Operators

2.8.1 `transpose`

The transpose of a matrix A can be written as

```
AT = transpose(A)
AT = A.T
AT = as_matrix(A[i,j], (j,i))
```

The definition of the transpose is

$$\texttt{AT[i,j]} \leftrightarrow (\mathbf{A}^\top)_{ij} = \mathbf{A}_{ji}. \tag{2.15}$$

For transposing higher order tensor expressions, index notation can be used:

```
AT = as_tensor(A[i,j,k,l], (l,k,j,i))
```

2.8.2 `tr`

The trace of a matrix A is the sum of the diagonal entries. This can be written as

```
t = tr(A)
t = A[i,i]
```

The definition of the trace is

$$\texttt{tr}(\mathrm{A}) \leftrightarrow \mathrm{tr}\mathbf{A} = A_{ii} = \sum_{i=0}^{n-1} A_{ii}. \tag{2.16}$$

2.8.3 `dot`

The dot product of two tensors a and b can be written

```
# General tensors
f = dot(a, b)

# Vectors a and b
f = a[i]*b[i]

# Matrices a and b
f = as_matrix(a[i,k]*b[k,j], (i,j))
```

The definition of the dot product of unit vectors is[3]

$$\mathbf{i}_i \cdot \mathbf{i}_j = \delta_{ij} \tag{2.17}$$

where δ_{ij} is the Kronecker delta as explained earlier. The dot product of higher order tensors follow from this, as illustrated with the following examples.

An example with two vectors

$$\mathbf{v} \cdot \mathbf{u} = (v_i \mathbf{i}_i) \cdot (u_j \mathbf{i}_j) = v_i u_j (\mathbf{i}_i \cdot \mathbf{i}_j) = v_i u_j \delta_{ij} = v_i u_i \tag{2.18}$$

[3] Assuming an orthonormal basis for a Euclidean space.

An example with a tensor of rank two

$$\mathbf{A} \cdot \mathbf{B} = (A_{ij}\mathbf{i}_i\mathbf{i}_j) \cdot (B_{kl}\mathbf{i}_k\mathbf{i}_l) \tag{2.19}$$
$$= (A_{ij}B_{kl})\mathbf{i}_i(\mathbf{i}_j \cdot \mathbf{i}_k)\mathbf{i}_l \tag{2.20}$$
$$= (A_{ij}B_{kl}\delta_{jk})\mathbf{i}_i\mathbf{i}_l \tag{2.21}$$
$$= A_{ik}B_{kl}\mathbf{i}_i\mathbf{i}_l. \tag{2.22}$$

This is the same as to matrix-matrix multiplication.

An example with a vector and a tensor of rank two

$$\mathbf{v} \cdot \mathbf{A} = (v_j\mathbf{i}_j) \cdot (A_{kl}\mathbf{i}_k\mathbf{i}_l) \tag{2.23}$$
$$= (v_jA_{kl})(\mathbf{i}_j \cdot \mathbf{i}_k)\mathbf{i}_l \tag{2.24}$$
$$= (v_jA_{kl}\delta_{jk})\mathbf{i}_l \tag{2.25}$$
$$= v_kA_{kl}\mathbf{i}_l \tag{2.26}$$

This is the same as to vector-matrix multiplication.

This generalizes to tensors of arbitrary rank: The dot product applies to the last axis of a and the first axis of b. The tensor rank of the product is rank(a)+rank(b)-2.

2.8.4 `inner`

The inner product is a contraction over all axes of a and b, that is the sum of all componentwise products. The operands must have the exact same dimensions. For two vectors it is equivalent to the dot product.

If $\mathbf{A}$ and $\mathbf{B}$ are rank 2 tensors and $\mathcal{C}$ and $\mathcal{D}$ are rank 3 tensors their inner products are

$$\mathbf{A} : \mathbf{B} = A_{ij}B_{ij} \tag{2.27}$$
$$\mathcal{C} : \mathcal{D} = C_{ijk}D_{ijk} \tag{2.28}$$

Using **UFL** notation, the following pairs of declarations are equivalent

```
# Vectors
f = inner(a, b)
f = v[i]*b[i]

# Matrices
f = inner(A, B)
f = A[i,j]*B[i,j]

# Rank 3 tensors
f = inner(C, D)
f = C[i,j,k]*D[i,j,k]
```

2.8.5 `outer`

The outer product of two tensors a and b can be written

```
A = outer(a, b)
```

The general definition of the outer product of two tensors $\mathcal{C}$ of rank r and $\mathcal{D}$ of rank s is

$$\mathcal{C} \otimes \mathcal{D} = C_{\iota_0^a \ldots \iota_{r-1}^a} D_{\iota_0^b \ldots \iota_{s-1}^b} \mathbf{i}_{\iota_0^a} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-2}^a} \otimes \mathbf{i}_{\iota_1^b} \otimes \cdots \otimes \mathbf{i}_{\iota_{s-1}^b} \tag{2.29}$$

Some examples with vectors and matrices are easier to understand

$$\mathbf{v} \otimes \mathbf{u} = v_i u_j \mathbf{i}_i \mathbf{i}_j, \tag{2.30}$$

$$\mathbf{v} \otimes \mathbf{B} = v_i B_{kl} \mathbf{i}_i \mathbf{i}_k \mathbf{i}_l, \tag{2.31}$$

$$\mathbf{A} \otimes \mathbf{B} = A_{ij} B_{kl} \mathbf{i}_i \mathbf{i}_j \mathbf{i}_k \mathbf{i}_l. \tag{2.32}$$

The outer product of vectors is often written simply as

$$\mathbf{v} \otimes \mathbf{u} = \mathbf{v}\mathbf{u}, \tag{2.33}$$

which is what we've done with $\mathbf{i}_i \mathbf{i}_j$ above.

The rank of the outer product is the sum of the ranks of the operands.

2.8.6 `cross`

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:

$$\texttt{cross(v, w)} \leftrightarrow \mathbf{v} \times \mathbf{w} = (v_1 w_2 - v_2 w_1, v_2 w_0 - v_0 w_2, v_0 w_1 - v_1 w_0). \quad (2.34)$$

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

2.8.7 `det`

The determinant of a matrix A can be written

```
d = det(A)
```

2.8.8 `dev`

The deviatoric part of matrix A can be written

```
B = dev(A)
```

2.8.9 `sym`

The symmetric part of A can be written

```
B = sym(A)
```

The definition is

$$sym\,\mathbf{A} = \frac{1}{2}(\mathbf{A} + \mathbf{A}^T) \quad (2.35)$$

2.8.10 `skew`

The skew symmetric part of A can be written

```
B = skew(A)
```

The definition is

$$skew\,\mathbf{A} = \frac{1}{2}(\mathbf{A} - \mathbf{A}^T) \tag{2.36}$$

2.8.11 `cofac`

The cofactor of a matrix A can be written

```
B = cofac(A)
```

The definition is

$$cofac\,\mathbf{A} = det(\mathbf{A})\mathbf{A}^{-1} \tag{2.37}$$

The implementation of this is currently rather crude, with a hardcoded symbolic expression for the cofactor. Therefore, this is limited to 1x1, 2x2 and 3x3 matrices.

2.8.12 `inv`

The inverse of matrix A can be written

```
Ainv = inv(A)
```

The implementation of this is currently rather crude, with a hardcoded symbolic expression for the inverse. Therefore, this is limited to 1x1, 2x2 and 3x3 matrices.

2.9 Differential Operators

Three different kinds of derivatives are currently supported: spatial derivatives, derivatives w.r.t. user defined variables, and derivatives of a form or functional w.r.t. a function.

2.9.1 Basic spatial derivatives

Spatial derivatives hold a special place in partial differential equations from physics and there are several ways to express those. The basic way is

```
# Derivative w.r.t. x_2
f = Dx(v, 2)
f = v.dx(2)
# Derivative w.r.t. x_i
g = Dx(v, i)
g = v.dx(i)
```

If `v` is a scalar expression, `f` here is the scalar derivative of `v` w.r.t. spatial direction z. If `v` has no free indices, `g` is the scalar derivative w.r.t. spatial direction x_i, and `g` has the free index `i`. Written as formulas, this can be expressed compactly using the $v_{,i}$ notation:

$$f = \frac{\partial v}{\partial x_2} = v_{,2}, \tag{2.38}$$

$$g = \frac{\partial v}{\partial x_i} = v_{,i}. \tag{2.39}$$

Note the resemblance of $v_{,i}$ and `v.dx(i)`.

If the expression to be differentiated w.r.t. x_i has `i` as a free index, implicit summation is implied.

```
# Sum of derivatives w.r.t. x_i for all i
g = Dx(v[i], i)
```

```
g = v[i].dx(i)
```

Here `g` will represent the sum of derivatives w.r.t. x_i for all `i`, that is

$$g = \sum_i \frac{\partial v}{\partial x_i} = v_{i,i}.$$

Note the compact index notation $v_{i,i}$ with implicit summation.

2.9.2 Compound spatial derivatives

UFL implements several common differential operators. The notation is simple and their names should be self explaining:

```
Df = grad(f)
df = div(f)
cf = curl(v)
rf = rot(f)
```

The operand `f` can have no free indices.

2.9.3 Gradient

The gradient of a scalar u is defined as

$$grad(u) \equiv \nabla u = \sum_{k=0}^{d-1} \frac{\partial u}{\partial x_k} \mathbf{i}_k, \tag{2.40}$$

which is a vector of all spatial partial derivatives of u.

The gradient of a vector $\mathbf{v}$ is defined as

$$grad(\mathbf{v}) \equiv \nabla \mathbf{v} = \frac{\partial v_i}{\partial x_j} \mathbf{i}_i \mathbf{i}_j, \tag{2.41}$$

which written componentwise is

$$\mathbf{A} = \nabla \mathbf{v}, \qquad A_{ij} = v_{i,j} \tag{2.42}$$

In general for a tensor $\mathbf{A}$ of rank r the definition is

$$grad(\mathbf{A}) \equiv \nabla \mathbf{A} = (\frac{\partial}{\partial x_i})(A_\iota \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}}) \otimes \mathbf{i}_i = \frac{\partial A_\iota}{\partial x_i} \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}} \otimes \mathbf{i}_i, \tag{2.43}$$

where ι is a multiindex of length r.

In **UFL**, the following pairs of declarations are equivalent:

```
Dfi = grad(f)[i]
Dfi = f.dx(i)

Dvi = grad(v)[i, j]
Dvi = v[i].dx(j)

DAi = grad(A)[..., i]
DAi = A.dx(i)
```

for a scalar expression `f`, a vector expression `v`, and a tensor expression `A` of arbitrary rank.

2.9.4 Divergence

The divergence of any nonscalar (vector or tensor) expression $\mathbf{A}$ is defined as the contraction of the partial derivative over the last axis of the expression.

TODO: Detailed examples like for gradient.

In **UFL**, the following declarations are equivalent:

```
dv = div(v)
dv = v[i].dx(i)
```

```
dA = div(A)
dA = A[..., i].dx(i)
```

for a vector expression v and a tensor expression A.

2.9.5 Curl and rot

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

$$\texttt{curl(v)} \leftrightarrow \operatorname{curl} \mathbf{v} = \nabla \times \mathbf{v} = (\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}). \quad (2.44)$$

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

2.9.6 Variable derivatives

UFL also supports differentiation with respect to user defined variables. A user defined variable can be any[4] expression that is defined as a variable.

The notation is illustrated here:

```
# Define some arbitrary expression
u = Function(element)
w = sin(u**2)

# Annotate expression w as a variable that can be used in diff
w = variable(w)

# This expression is a function of w
F = I + diff(u, x)
```

[4] TODO: There are probably some things that don't make sense.

```
# The derivative of expression f w.r.t. the variable w
df = diff(f, w)
```

Note that the variable `w` still represents the same expression.

This can be useful for example to implement material laws in hyperelasticity where the stress tensor is derived from a Helmholtz strain energy function.

Currently, **UFL** does not implement time in any particular way, but differentiation w.r.t. time can be done without this support through the use of a constant variable t:

```
t = variable(Constant(cell))
f = sin(x[0])**2 * cos(t)
dfdt = diff(f, t)
```

2.9.7 Functional derivatives

The third and final kind of derivatives are derivatives of functionals or forms w.r.t. to a `Function`. This is described in more detail in section 2.13.6 about form transformations.

2.10 DG operators

UFL 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.

2.10.1 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.

2.10.2 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:

$$\texttt{jump(v)} \leftrightarrow [\![v]\!] = v^+ - v^-. \tag{2.45}$$

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

$$\texttt{jump(v, n)} \leftrightarrow [\![v]\!]_n = v^+ n^+ + v^- n^-. \tag{2.46}$$

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

$$\texttt{jump(v, n)} \leftrightarrow [\![v]\!]_n = v^+ \cdot n^+ + v^- \cdot n^-. \tag{2.47}$$

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.

2.10.3 Average: `avg(v)`

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

$$\text{avg(v)} \leftrightarrow \langle v \rangle = \frac{1}{2}(v^+ + v^-). \tag{2.48}$$

The expression `avg(v)` has the same value shape as the expression `v`.

2.11 Conditional Operators

2.11.1 Conditional

UFL has limited support for branching, but for some PDEs it is needed. The expression `c` in

```
c = conditional(condition, true_value, false_value)
```

evaluates to `true_value` at run-time if `condition` evaluates to true, or to `false_value` otherwise.

This corresponds to the C++ syntax `(condition ? true_value: false_value)`, or the Python syntax `(true_value if condition else false_value)`,

2.11.2 Conditions

- `eq(a, b)` represents the condition that a == b
- `ne(a, b)` represents the condition that a != b
- `le(a, b)` represents the condition that a ¡= b
- `ge(a, b)` represents the condition that a ¿= b
- `lt(a, b)` represents the condition that a ¡ b
- `gt(a, b)` represents the condition that a ¿ b

TODO: This is rather limited, probably need the operations "and" and "or" as well, the syntax will be rather convoluted... Can we improve? Low priority though.

[**Advanced**] Because of details in the way Python behaves, we cannot overload the builtin comparison operators for this purpose, hence these named operators.

2.12 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

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top). \tag{2.49}$$

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

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

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

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

2.13 Form Transformations

When you have defined a `Form`, you can derive new related forms from it automatically. UFL defines a set of common form transformations described in this section.

2.13.1 Replacing arguments of a Form

The function `replace` lets you replace terminal objects with other values, using a mapping defined by a Python dict. This can be used for example to replace a `Function` with a fixed value for optimized runtime evaluation.

```
f = Function(element)
g = Function(element)
c = Constant(cell)
a = f*g*v*dx
b = replace(a, { f: 3.14, g: c })
```

The replacement values must have the same basic properties as the original values, in particular value shape and free indices.

2.13.2 Action of a form on a function

The action of a bilinear form a is defined as

$$b(v; w) = a(v, w),$$

The action of a linear form L is defined as

$$f(;w) = L(w)$$

This operation is implemented in UFL simply by replacing the rightmost basis function (trial function for a, test function for L) in a `Form`, and is used like this:

```
L = action(a, w)
f = action(L, w)
```

To give a concrete example, these declarations are equivalent:

```
a = inner(grad(u), grad(v))*dx
L = action(a, w)

a = inner(grad(u), grad(v))*dx
L = inner(grad(w), grad(v))*dx
```

If a is a rank 2 form used to assemble the matrix A, L is a rank 1 form that can be used to assemble the vector $b = Ax$ directly. This can be used to define both the form of a matrix and the form of its action without code duplication, and for the action of a Jacobi matrix computed using derivative.

If L is a rank 1 form used to assemble the vector b, f is a functional that can be used to assemble the scalar value $f = b \cdot w$ directly. This operation is sometimes used in, e.g., error control with L being the residual equation and w being the solution to the dual problem. (However, the discrete vector for the assembled residual equation will typically be available, so doing the dot product using linear algebra would be faster than using this feature.) FIXME: Is this right?

2.13.3 Energy norm of a bilinear Form

The functional representing the energy norm $|v|_A = v^T A v$ of a matrix A assembled from a form a can be computed like this

```
f = energy_norm(a, w)
```

which is equivalent to

```
f = action(action(a, w), w)
```

2.13.4 Adjoint of a bilinear Form

The adjoint a' of a bilinear form a is defined as

$$a'(u, v) = a(v, u).$$

This operation is implemented in UFL simply by swapping test and trial functions in a `Form`, and is used like this:

```
aprime = adjoint(a)
```

2.13.5 Linear and bilinear parts of a Form

Some times it is useful to write an equation on the format

$$a(v, u) - L(v) = 0.$$

Before we can assemble the linear equation

$$Au = b,$$

we need to extract the forms corresponding to the left hand side and right hand side. This corresponds to extracting the bilinear and linear terms of the form respectively, or the terms that depend on both a test and a trial function on one side and the terms that depend on only a test function on the other.

This is easily done in UFL using `lhs` and `rhs`:

```
b = u*v*dx - f*v*dx
a, L = lhs(b), rhs(b)
```

Note that `rhs` multiplies the extracted terms by -1, corresponding to moving them from left to right, so this is equivalent to

```
a = u*v*dx
L = f*v*dx
```

As a slightly more complicated example, this formulation

```
F = v*(u - w)*dx + k*dot(grad(v), grad(0.5*(w + u)))*dx
a, L = lhs(F), rhs(F)
```

is equivalent to

```
a = v*u*dx + k*dot(grad(v), 0.5*grad(u))*dx
L = v*w*dx - k*dot(grad(v), 0.5*grad(w))*dx
```

2.13.6 Automatic Functional Differentiation

UFL can compute derivatives of functionals or forms w.r.t. to a `Function`. This functionality can be used for example to linearize your nonlinear residual equation automatically, or derive a linear system from a functional, or compute sensitivity vectors w.r.t. some coefficient.

A functional can be differentiated to obtain a linear form,

$$F(v; w) = \frac{d}{dw} f(; w)$$

and a linear form [5] can be differentiated to obtain the bilinear form corresponding to its Jacobi matrix:

$$J(v, u; w) = \frac{d}{dw} F(v; w).$$

The UFL code to express this is (for a simple functional $f(w) = \int_\Omega \frac{1}{2} w^2 \, dx$)

```
f = (w**2)/2 * dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

which is equivalent to:

```
f = (w**2)/2 * dx
F = w*v*dx
J = u*v*dx
```

Assume in the following examples that:

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

The stiffness matrix can be computed from the functional $\int_\Omega \nabla w : \nabla w \, dx$, by the lines

```
f = inner(grad(w), grad(w))/2 * dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

[5] Note that by "linear form" we only mean a form that is linear in its test function, not in the function you differentiate with respect to.

which is equivalent to:

```
f = inner(grad(w), grad(w))/2 * dx
F = inner(grad(w), grad(v)) * dx
J = inner(grad(u), grad(v)) * dx
```

Note that here the basis functions are provided explicitly, which is some times necessary, e.g., if part of the form is linearlized manually like in (*TODO: An example that makes sense would be nicer, this is just a random form.*)

```
g = Function(element)
f = inner(grad(w), grad(w))*dx
F = derivative(f, w, v) + dot(w-g,v)*dx
J = derivative(F, w, u)
```

Derivatives can also be computed w.r.t. functions in mixed spaces. Consider this example, an implementation of the harmonic map equations using automatic differentiation.

```
X = VectorElement("Lagrange", cell, 1)
Y = FiniteElement("Lagrange", cell, 1)

x = Function(X)
y = Function(Y)

L = inner(grad(x), grad(x))*dx + dot(x,x)*y*dx

F = derivative(L, (x,y))
J = derivative(F, (x,y))
```

Here `L` is defined as a functional with two coefficient functions `x` and `y` from separate finite element spaces. However, `F` and `J` become linear and bilinear forms respectively with basis functions defined on the mixed finite element

```
M = X + Y
```

There is a subtle difference between defining `x` and `y` separately and this alternative implementation (reusing the elements `X`,`Y`,`M`):

```
u = Function(M)
x, y = split(u)

L = inner(grad(x), grad(x))*dx + dot(x,x)*y*dx

F = derivative(L, u)
J = derivative(F, u)
```

The difference is that the forms here have *one* coefficient function `u` in the mixed space, and the forms above have *two* coefficient functions `x` and `y`.

TODO: Move this to implementation part? If you wonder how this is all done, a brief explanation follows. Recall that a `Function` represents a sum of unknown coefficients multiplied with unknown basis functions in some finite element space.

$$w(x) = \sum_k w_k \phi_k(x) \tag{2.50}$$

Also recall that a `BasisFunction` represents any (unknown) basis function in some finite element space.

$$v(x) = \phi_k(x), \qquad \phi_k \in V_h. \tag{2.51}$$

A form $L(v; w)$ implemented in **UFL** is intended for discretization like

$$b_i = L(\phi_i; \sum_k w_k \phi_k), \qquad \forall \phi_i \in V_h. \tag{2.52}$$

The Jacobi matrix A_{ij} of this vector can be obtained by differentiation of b_i w.r.t. w_j, which can be written

$$A_{ij} = \frac{db_i}{dw_j} = a(\phi_i, \phi_j; \sum_k w_k \phi_k), \qquad \forall \phi_i \in V_h, \quad \forall \phi_j \in V_h, \tag{2.53}$$

for some form a. In **UFL**, the form a can be obtained by differentiating L. To manage this, we note that as long as the domain Ω is independent of w_j, $\int_\Omega$ commutes with $\frac{d}{dw_j}$, and we can differentiate the integrand expression instead, e.g.,

$$L(v; w) = \int_\Omega I_c(v; w)\, dx + \int_{\partial\Omega} I_e(v; w)\, ds, \tag{2.54}$$

$$\frac{d}{dw_j} L(v; w) = \int_\Omega \frac{dI_c}{dw_j}\, dx + \int_{\partial\Omega} \frac{dI_e}{dw_j}\, ds. \tag{2.55}$$

In addition, we need that

$$\frac{dw}{dw_j} = \phi_j, \qquad \forall \phi_j \in V_h, \tag{2.56}$$

which in **UFL** can be represented as

$$w = \texttt{Function(element)}, \tag{2.57}$$

$$v = \texttt{BasisFunction(element)}, \tag{2.58}$$

$$\frac{dw}{dw_j} = v, \tag{2.59}$$

since w represents the sum and v represents any and all basis functions in V_h.

Other operators have well defined derivatives, and by repeatedly applying the chain rule we can differentiate the integrand automatically.

The notation here has potential for improvement, feel free to ask if something is unclear, or suggest improvements.

2.13.7 Combining form transformations

Form transformations can be combined freely. Note that to do this, derivatives are usually be evaluated before applying e.g. the action of a form, because `derivative` changes the arity of the form.

```
element = FiniteElement("CG", cell, 1)
w = Function(element)
```

```
f = w**4/4*dx(0) + inner(grad(w), grad(w))*dx(1)
F = derivative(f, w)
J = derivative(F, w)
Ja = action(J, w)
Jp = adjoint(J)
Jpa = action(Jp, w)
g = Function(element)
Jnorm = energy_norm(J, g)
```

TODO: Find some more examples, e.g. from error control!

2.14 Tuple Notation

In addition to the standard integrand notation described above, UFL supports a simplified *tuple notation* by which L^2 inner products may be expressed as tuples. Consider for example the following bilinear form as part of a variational problem for a reaction–diffusion problem:

$$\begin{aligned} a(v, u) &= \int_\Omega \nabla v \cdot \nabla u + vu \,\mathrm{d}x \\ &= (\nabla v, \nabla u) + (v, u) \end{aligned}$$

In standard UFL notation, this bilinear form may be expressed as

```
a = inner(grad(v), grad(u))*dx + v*u*dx
```

In tuple notation, this may alternatively be expressed as

```
a = (grad(v), grad(u)) + (v, u)
```

In general, a form may be expressed as a sum of tuples or triples of the form

```
(v, w)
(v, w, dm)
```

where `v` and `w` are expressions of matching rank (so that `inner(v, w)` makes sense), and `dm` is a measure. If the measure is left out, it is assumed that it is `dx`.

The following example illustrates how to express a form containing integrals over subdomains and facets:

```
a = (grad(v), grad(u)) + (v, b*grad(u), dx(2))
  + (v, u, ds) + (jump(v), jump(u), dS)
```

The following caveats should be noted:

- The only operation allowed on a tuple is addition. In particular, tuples may not subtracted. Thus, `a = (grad(v), grad(u)) - (v, u)` must be expressed as `a = (grad(v), grad(u)) + (-v, u)`.
- Tuple notation may not be mixed with standard UFL integrand notation. Thus, `a = (grad(v), grad(u)) + inner(v, u)*dx` is not valid.

[**Advanced**] Tuple notation is strictly speaking not a part of the form language, but tuples may be converted to UFL forms using the function `tuple2form` available from the module `ufl.algorithms`. This is normally handled automatically by form compilers, but the `tuple2form` utility may useful when working with UFL from a Python script. Automatic conversion is also carried out by UFL form operators such as `lhs` and `rhs`.

2.15 Form Files

UFL forms and elements can be collected in a *form file* with the extension `.ufl`. Form compilers will typically execute this file with the global **UFL**

namespace available, and extract forms and elements that are defined after execution. The compilers do not compile all forms and elements that are defined in file, but only those that are *exported*. A finite element with the variable name `element` is exported by default, as are forms with the names `M`, `L`, and `a`. The default form names are intended for a functional, linear form, and bilinear form respectively.

To export multiple forms and elements or use other names, an explicit list with the forms and elements to export can be defined. Simply write

```
elements = [V, P, TH]
forms = [a, L, F, J, L2, H1]
```

at the end of the file to export the elements and forms held by these variables.

Chapter 3

Example Forms

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 `demo/` of the **UFL** source tree together with numerous other examples.

3.1 The mass matrix

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

$$a(v, u) = \int_\Omega v\, u \,\mathrm{d}x, \tag{3.1}$$

which can be implemented in **UFL** 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.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.2 Poisson's equation

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

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x, \tag{3.2}$$

$$L(v; f) = \int_\Omega v \, f \, \mathrm{d}x, \tag{3.3}$$

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 like this:

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

or like this:

```
a = v.dx(i)*u.dx(i)*dx
```

This example is implemented in the file `poisson.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.3 Vector-valued Poisson

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

$$a(v, u) = \int_\Omega \nabla v : \nabla u \, \mathrm{d}x, \tag{3.4}$$

$$L(v; f) = \int_\Omega v \cdot f \, \mathrm{d}x, \tag{3.5}$$

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

Alternatively, index notation may be used like this:

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

or like this:

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

This example is implemented in the file `poisson_system.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.4 The strain-strain term of linear elasticity

The strain-strain term of linear elasticity,

$$a(v, u) = \int_\Omega \epsilon(v) : \epsilon(u) \, \mathrm{d}x, \tag{3.6}$$

where

$$\epsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^\top) \tag{3.7}$$

can be implemented as follows:

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

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

def epsilon(v):
    Dv = grad(v)
    return 0.5*(Dv + Dv.T)

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

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

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

or like this:

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

This example is implemented in the file `elasticity.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.5 The nonlinear term of Navier–Stokes

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

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

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

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

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

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

alternatively using index notation like this:

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

or like this:

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

This example is implemented in the file `navier_stokes.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.6 The heat equation

Discretizing the heat equation,

$$\dot{u} - \nabla \cdot (c \nabla u) = f, \tag{3.9}$$

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

$$\frac{1}{k_n} \int_\Omega v \, (u_h^n - u_h^{n-1}) \, \mathrm{d}x + \int_\Omega c \, \nabla v \cdot \nabla u_h^n \, \mathrm{d}x = \int_\Omega v \, f^n \, \mathrm{d}x \tag{3.10}$$

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:

$$a(v, u_h^n; c, k) = \int_\Omega v \, u_h^n \, \mathrm{d}x + k_n \int_\Omega c \, \nabla v \cdot \nabla u_h^n \, \mathrm{d}x, \tag{3.11}$$

$$L(v; u_h^{n-1}, f, k) = \int_\Omega v \, u_h^{n-1} \, \mathrm{d}x + k_n \int_\Omega v \, f^n \, \mathrm{d}x, \tag{3.12}$$

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("triangle")   # Time step

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

This example is implemented in the file `heat.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.7 Mixed formulation of Stokes

To solve Stokes' equations,

$$-\Delta u + \nabla p = f, \tag{3.13}$$
$$\nabla \cdot u = 0, \tag{3.14}$$

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:

$$a((v, q), (u, p)) = \int_\Omega \nabla v : \nabla u - (\nabla \cdot v)\, p + q\, (\nabla \cdot u)\, \mathrm{d}x, \tag{3.15}$$
$$L((v, q); f) = \int_\Omega v \cdot f\, \mathrm{d}x. \tag{3.16}$$

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

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

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

f = Function(P2)

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

This example is implemented in the file `stokes.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.8 Mixed formulation of Poisson

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

$$\sigma + \nabla u = 0, \tag{3.17}$$

$$\nabla \cdot \sigma = f. \tag{3.18}$$

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 $(\sigma, u) \in V = H(\mathrm{div}) \times L_2$ such that

$$a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall\, (\tau, w) \in V, \tag{3.19}$$

where

$$a((\tau, w), (\sigma, u)) = \int_\Omega \tau \cdot \sigma - \nabla \cdot \tau\, u + w \nabla \cdot \sigma \,\mathrm{d}x, \tag{3.20}$$

$$L((\tau, w); f) = \int_\Omega w \cdot f \,\mathrm{d}x. \tag{3.21}$$

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

```
cell = triangle
BDM1 = FiniteElement("Brezzi-Douglas-Marini", cell, 1)
DG0  = FiniteElement("Discontinuous Lagrange", cell, 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 `mixed_poisson.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.9 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

$$\begin{aligned} a(v, u; h) = &\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x \\ &+ \sum_S \int_S -\langle \nabla v \rangle \cdot [\![u]\!]_n - [\![v]\!]_n \cdot \langle \nabla u \rangle + (\alpha/h)[\![v]\!]_n \cdot [\![u]\!]_n \,\mathrm{d}S \\ &+ \int_{\partial\Omega} -\nabla v \cdot [\![u]\!]_n - [\![v]\!]_n \cdot \nabla u + (\gamma/h) v u \,\mathrm{d}s \\ L(v; f, g) = &\int_\Omega v f \,\mathrm{d}x + \int_{\partial\Omega} v g \,\mathrm{d}s. \end{aligned} \tag{3.22}$$

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

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

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

f = Function(DG1)
g = Function(DG1)
#h = MeshSize(cell) # TODO: Do we include MeshSize in UFL?
h = Constant(cell)
alpha = 1 # TODO: Set to proper value
gamma = 1 # TODO: Set to proper value

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

This example is implemented in the file `poisson_dg.ufl` in the collection of demonstration forms included with the **UFL** source distribution.

3.10 Quadrature elements

FIXME: The code examples in this section have been mostly converted to **UFL** *syntax, but the quadrature elements need some more updating, as well as the text. In UFL, I think we should define the element order and not the*

number of points for quadrature elements, and let the form compiler choose a quadrature rule. This way the form depends less on the cell in use.

We consider here a nonlinear version of the Poisson's equation to illustrate the main point of the `"Quadrature"` finite element family. The strong equation looks as follows:

$$-\nabla \cdot (1 + u^2)\nabla u = f. \tag{3.23}$$

The linearised bilinear and linear forms for this equation,

$$a(v, u; u_0) = \int_\Omega (1 + u_0^2)\nabla v \cdot \nabla u \,\mathrm{d}x + \int_\Omega 2u_0 u \nabla v \cdot \nabla u_0 \,\mathrm{d}x, \tag{3.24}$$

$$L(v; u_0, f) = \int_\Omega v\, f \,\mathrm{d}x - \int_\Omega (1 + u_0^2)\nabla v \cdot \nabla u_0 \,\mathrm{d}x, \tag{3.25}$$

can be implemented in a single form file as follows:

```
# NonlinearPoisson.ufl
element = FiniteElement("Lagrange", triangle, 1)

v = TestFunction(element)
u = TrialFunction(element)
u0 = Function(element)
f = Function(element)

a = (1+u0**2)*dot(grad(v), grad(u))*dx \
  + 2*u0*u*dot(grad(v), grad(u0))*dx
L = v*f*dx - (1+u0**2)*dot(grad(v), grad(u0))*dx
```

Here, u_0 represents the solution from the previous Newton-Raphson iteration.

The above form will be denoted REF1 and serve as our reference implementation for linear elements. A similar form (REF2) using quadratic elements will serve as a reference for quadratic elements.

Now, assume that we want to treat the quantities $C = (1 + u_0^2)$ and $\sigma_0 = (1+u_0^2)\nabla u_0$ as given functions (to be computed elsewhere). Substituting into

bilinear linear forms, we obtain

$$a(v,u) = \int_\Omega \mathrm{C}\nabla v \cdot \nabla u \,\mathrm{d}x + \int_\Omega 2u_0 u \nabla v \cdot \nabla u_0 \,\mathrm{d}x, \tag{3.26}$$

$$L(v;\sigma_0,f) = \int_\Omega v\, f \,\mathrm{d}x - \int_\Omega \nabla v \cdot \sigma_0 \,\mathrm{d}x. \tag{3.27}$$

Then, two additional forms are created to compute the tangent C and the gradient of u_0. This situation shows up in plasticity and other problems where certain quantities need to be computed elsewhere (in user-defined functions). The 3 forms using the standard `FiniteElement` (linear elements) can then be implemented as:

```
# FE1NonlinearPoisson.ufl
element = FiniteElement("Lagrange", triangle, 1)
DG = FiniteElement("Discontinuous Lagrange", triangle, 0)
sig = VectorElement("Discontinuous Lagrange", triangle, 0)

v    = TestFunction(element)
u    = TrialFunction(element)
u0   = Function(element)
C    = Function(DG)
sig0 = Function(sig)
f    = Function(element)

a = v.dx(i)*C*u.dx(i)*dx + v.dx(i)*2*u0*u*u0.dx(i)*dx
L = v*f*dx - dot(grad(v), sig0)*dx
```

```
# FE1Tangent.ufl
element = FiniteElement("Lagrange", triangle, 1)
DG = FiniteElement("Discontinuous Lagrange", triangle, 0)

v = TestFunction(DG)
u = TrialFunction(DG)
u0= Function(element)

a = v*u*dx
L = v*(1.0 + u0**2)*dx
```

```
# FE1Gradient.ufl
element = FiniteElement("Lagrange", triangle, 1)
DG = VectorElement("Discontinuous Lagrange", triangle, 0)

v = TestFunction(DG)
u = TrialFunction(DG)
u0 = Function(element)

a = dot(v, u)*dx
L = dot(v, grad(u0))*dx
```

The 3 forms can be implemented using the `QuadratureElement` in a similar fashion in which only the element declaration is different:

```
# QE1NonlinearPoisson.ufl
element = FiniteElement("Lagrange", triangle, 1)
QE = FiniteElement("Quadrature", triangle, 2)
sig = VectorElement("Quadrature", triangle, 2)
```

```
# QE1Tangent.ufl
element = FiniteElement("Lagrange", triangle, 1)
QE = FiniteElement("Quadrature", triangle, 2)
```

```
# QE1Gradient.ufl
element = FiniteElement("Lagrange", triangle, 1)
QE = VectorElement("Quadrature", triangle, 2)
```

Note that we use 2 points when declaring the `QuadratureElement`. This is because the RHS of the `Tangent.form` is 2^{nd} order and therefore we need 2 points for exact integration. Due to consistency issues, when passing functions around between the forms, we also need to use 2 points when declaring the `QuadratureElement` in the other forms.

Typical values of the relative residual for each Newton iteration for all 3 approaches are shown in Table 3.1. It is noted that the convergence rate is quadratic as it should be for all 3 methods.

Iteration	REF1	FE1	QE1
1	6.342e-02	6.342e-02	6.342e-02
2	5.305e-04	5.305e-04	5.305e-04
3	3.699e-08	3.699e-08	3.699e-08
4	2.925e-16	2.925e-16	2.475e-16

Table 3.1: Relative residuals for each approach for linear elements.

However, if quadratic elements are used to interpolate the unknown field u, the order of all elements in the above forms is increased by 1. This influences the convergence rate as seen in Table 3.2. Clearly, using the standard `FiniteElement` leads to a poor convergence whereas the `QuadratureElement` still leads to quadratic convergence.

Iteration	REF2	FE2	QE2
1	2.637e-01	3.910e-01	2.644e-01
2	1.052e-02	4.573e-02	1.050e-02
3	1.159e-05	1.072e-02	1.551e-05
4	1.081e-11	7.221e-04	9.076e-09

Table 3.2: Relative residuals for each approach for quadratic elements.

3.11 More Examples

Feel free to send additional demo form files for your favourite PDE to the UFL mailing list.

Chapter 4

Internal Representation Details

This chapter explains how **UFL** forms and expressions are represented in detail. Most operations are mirrored by a representation class, e.g., `Sum` and `Product`, all which are subclasses of `Expr`. You can import all of them from the submodule `ufl.classes` by

```
from ufl.classes import *
```

TODO: Automate the construction of class hierarchy figures using ptex2tex.

4.1 Structure of a Form

TODO: Add class relations figure with Form, Integral, Expr, Terminal, Operator.

Each `Form` owns multiple `Integral` instances, each associated with a different `Measure`. An `Integral` owns a `Measure` and an `Expr`, which represents the integrand expression. The `Expr` is the base class of all expressions. It has two direct subclasses `Terminal` and `Operator`.

Subclasses of `Terminal` represent atomic quantities which terminate the expression tree, e.g. they have no subexpressions. Subclasses of `Operator` represent operations on one or more other expressions, which may usually be `Expr` subclasses of arbitrary type. Different `Operator`s may have restrictions on some properties of their arguments.

All the types mentioned here are conceptually immutable, i.e. they should never be modified over the course of their entire lifetime. When a modified expression, measure, integral, or form is needed, a new instance must be created, possibly sharing some data with the old one. Since the shared data is also immutable, sharing can cause no problems.

4.2 General properties of expressions

Any **UFL** expression has certain properties, defined by functions that every `Expr` subclass must implement. In the following, `u` represents an arbitrary **UFL** expression, i.e. an instance of an arbitrary `Expr` subclass.

4.2.1 `operands`

`u.operands()` returns a tuple with all the operands of u, which should all be `Expr` instances.

4.2.2 `reconstruct`

`u.reconstruct(operands)` returns a new `Expr` instance representing the same operation as `u` but with other operands. Terminal objects may simply return `self` since all `Expr` instance are immutable. An important invariant is that `u.reconstruct(u.operands()) == u`.

4.2.3 `cell`

`u.cell()` returns the first `Cell` instance found in `u`. It is currently assumed in **UFL** that no two different cells are used in a single form. Not all expression define a cell, in which case this returns `None` and `u` is spatially constant. Note that this property is used in some algorithms.

4.2.4 `shape`

`u.shape()` returns a tuple of integers, which is the tensor shape of `u`.

4.2.5 `free_indices`

`u.free_indices()` returns a tuple of `Index` objects, which are the unassigned, free indices of `u`.

4.2.6 `index_dimensions`

`u.index_dimensions()` returns a `dict` mapping from each `Index` instance in `u.free_indices()` to the integer dimension of the value space each index can range over.

4.2.7 `str(u)`

`str(u)` returns a human-readable string representation of `u`.

4.2.8 `repr(u)`

`repr(u)` returns a Python string representation of `u`, such that `eval(repr(u)) == u` holds in Python.

4.2.9 `hash(u)`

`hash(u)` returns a hash code for `u`, which is used extensively (indirectly) in algorithms whenever `u` is placed in a Python `dict` or `set`.

4.2.10 `u == v`

`u == v` returns true if and only if `u` and `v` represents the same expression in the exact same way. This is used extensively (indirectly) in algorithms whenever `u` is placed in a Python `dict` or `set`.

4.2.11 About other relational operators

In general, **UFL** expressions are not possible to fully evaluate since the cell and the values of form arguments are not available. Implementing relational operators for immediate evaluation is therefore impossible.

Overloading relational operators as a part of the form language is not possible either, since it interferes with the correct use of container types in Python like `dict` or `set`.

4.3 Elements

All finite element classes have a common base class `FiniteElementBase`. The class hierarchy looks like this:

TODO: Class figure.

TODO: Describe all FiniteElementBase subclasses here.

4.4 Terminals

All `Terminal` subclasses have some non-`Expr` data attached to them. `ScalarValue` has a Python scalar, `Function` has a `FiniteElement`, etc.

Therefore, a unified implementation of `reconstruct` is not possible, but since all `Expr` instances are immutable, `reconstruct` for terminals can simply return self. This feature and the immutability property is used extensively in algorithms.

TODO: Describe all Terminal representation classes here.

4.5 Operators

All instances of `Operator` subclasses are fully specified by their type plus the tuple of `Expr` instances that are the operands. Their constructors should take these operands as the positional arguments, and only that. This way, a unified implementation of `reconstruct` is possible, by simply calling the constructor with new operands. This feature is used extensively in algorithms.

TODO: Describe all Operator representation classes here.

4.6 Extending UFL

Adding new types to the **UFL** class hierarchy must be done with care. If you can get away with implementing a new operator as a combination of existing ones, that is the easiest route. The reason is that only some of the properties of an operator is represented by the `Expr` subclass. Other properties are part of the various algorithms in **UFL**. One example is derivatives, which are defined in the differentiation algorithm, and how to render a type to the LaTeX or dot formats. These properties could be merged into the class hierarchy, but other properties like how to map a **UFL** type to some **FFC** or

SFC or **DOLFIN** type can not be part of **UFL**. So before adding a new class, consider that doing so may require changes in multiple algorithms and even other projects.

TODO: More issues to consider when adding stuff to ufl.

Chapter 5

Algorithms

Algorithms to work with **UFL** forms and expressions can be found in the submodule `ufl.algorithms`. You can import all of them with the line

```
from ufl.algorithms import *
```

This chapter gives an overview of (most of) the implemented algorithms. The intended audience is primarily developers, but advanced users may find information here useful for debugging.

While domain specific languages introduce notation to express particular ideas more easily, which can reduce the probability of bugs in user code, they also add yet another layer of abstraction which can make debugging more difficult when the need arises. Many of the utilities described here can be useful in that regard.

5.1 Formatting expressions

Expressions can be formatted in various ways for inspection, which is particularly useful for debugging. We use the following as an example form for

the formatting sections below:

```
element = FiniteElement("CG", triangle, 1)
v = TestFunction(element)
u = TrialFunction(element)
c = Function(element)
f = Function(element)
a = c*u*v*dx + f*v*ds
```

5.1.1 str

Compact human readable pretty printing. Useful in interactive Python sessions. Example output of `str(a)`:

```
TODO
```

5.1.2 repr

Accurate description of expression, with the property that `eval(repr(a)) == a`. Useful to see which representation types occur in an expression, especially if `str(a)` is ambiguous. Example output of `repr(a)`:

```
TODO
```

5.1.3 Tree formatting

Ascii tree formatting, useful to inspect the tree structure of an expression in interactive Python sessions. Example output of `tree_format(a)`:

```
TODO
```

5.1.4 LaTeX formatting

See chapter about commandline utilities.

5.1.5 Dot formatting

See chapter about commandline utilities.

5.2 Inspecting and manipulating the expression tree

This subsection is mostly for form compiler developers and technically interested users.

TODO: More details about traversal and transformation algorithms for developers.

5.2.1 Traversing expressions

`iter_expressions`

```
q = f*v
r = g*v
s = u*v
a = q*dx(0) + r*dx(1) + s*ds(0)
for e in iter_expressions(a):
    print str(e)
```

`post_traversal`

TODO: traversal.py

`pre_traversal`

TODO: traversal.py

`walk`

TODO: traversal.py

`traverse_terminals`

TODO: traversal.py

5.2.2 Extracting information

TODO: analysis.py

5.2.3 Transforming expressions

So far the algorithms presented has been about inspecting expressions in various ways. Some recurring patterns occur when writing algorithms to modify expressions, either to apply mathematical transformations or to change their representation. Usually, different expression node types need different treatment.

To assist in such algorithms, **UFL** provides the `Transformer` class. This implements a variant of the Visitor pattern to enable easy definition of trans-

formation rules for the types you wish to handle.

Shown here is maybe the simplest transformer possible:

```
class Printer(Transformer):
    def __init__(self):
        Transformer.__init__(self)

    def expr(self, o, *operands):
        print "Visiting", str(o), "with operands:"
        print ", ".join(map(str,operands))
        return o

element = FiniteElement("CG", triangle, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = u*v

p = Printer()
p.visit(a)
```

The call to `visit` will traverse `a` and call `Printer.expr` on all expression nodes in post–order, with the argument `operands` holding the return values from visits to the operands of `o`. The output is:

```
TODO
```

Implementing `expr` above provides a default handler for any expression node type. For each subclass of `Expr` you can define a handler function to override the default by using the name of the type in underscore notation, e.g. `basis_function` for `BasisFunction`. The constructor of `Transformer` and implementation of `Transformer.visit` handles the mapping from type to handler function automatically.

Here is a simple example to show how to override default behaviour:

```
class FunctionReplacer(Transformer):
    def __init__(self):
        Transformer.__init__(self)

    expr = Transformer.reuse_if_possible
    terminal = Transformer.always_reuse

    def function(self, o):
        return FloatValue(3.14)

element = FiniteElement("CG", triangle, 1)
v = TestFunction(element)
f = Function(element)
a = f*v

r = FunctionReplacer()
b = r.visit(a)
print b
```

The output of this code is the transformed expression `b == 3.14*v`. This code also demonstrates how to reuse existing handlers. The handler `Transformer.reuse_if_po` will return the input object if the operands have not changed, and otherwise reconstruct a new instance of the same type but with the new transformed operands. The handler `Transformer.always_reuse` always reuses the instance without recursing into its children, usually applied to terminals. To set these defaults with less code, inherit `ReuseTransformer` instead of `Transformer`. This ensures that the parts of the expression tree that are not changed by the transformation algorithms always reuse the same instances.

We have already mentioned the difference between pre–traversal and post–traversal, and some times you need to combine the two. `Transformer` makes this easy by checking the number of arguments to your handler functions to see if they take transformed operands as input or not. If a handler function does not take more than a single argument in addition to self, its children are not visited automatically, and the handler function must call `visit` on its operands itself.

Here is an example of mixing pre– and post–traversal:

```
class Traverser(ReuseTransformer):
    def __init__(self):
        ReuseTransformer.__init__(self)

    def sum(self, o):
        operands = o.operands()
        newoperands = []
        for e in operands:
            newoperands.append( self.visit(e) )
        return sum(newoperands)

element = FiniteElement("CG", triangle, 1)
f = Function(element)
g = Function(element)
h = Function(element)
a = f+g+h

r = Traverser()
b = r.visit(a)
print b
```

This code inherits the `ReuseTransformer` like explained above, so the default behaviour is to recurse into children first and then call `Transformer.reuse_if_possible` to reuse or reconstruct each expression node. Since `sum` only takes `self` and the expression node instance `o` as arguments, its children are not visited automatically, and `sum` calls on `self.visit` to do this explicitly.

5.3 Automatic differentiation implementation

This subsection is mostly for form compiler developers and technically interested users.

TODO: More details about AD algorithms for developers.

5.3.1 Forward mode

TODO: forward_ad.py

5.3.2 Reverse mode

TODO: reverse_ad.py

5.3.3 Mixed derivatives

TODO: ad.py

5.4 Computational graphs

This section is for form compiler developers and is probably of no interest to end-users.

An expression tree can be seen as a directed acyclic graph (DAG). To aid in the implementation of form compilers, UFL includes tools to build a linearized[1] computational graph from the abstract expression tree.

A graph can be partitioned into subgraphs based on dependencies of subexpressions, such that a quadrature based compiler can easily place subexpressions inside the right sets of loops.

5.4.1 The computational graph

TODO: finish graph.py

[1] Linearized as in a linear datastructure, do not confuse this with automatic differentiation.

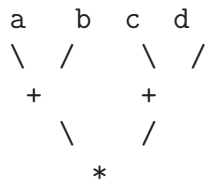
TODO

Consider the expression

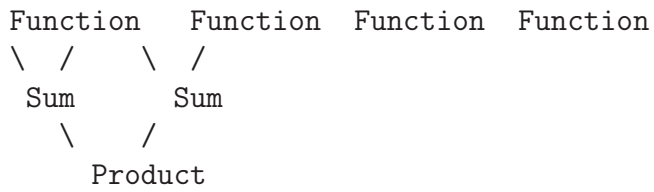
$$f = (a + b) * (c + d) \tag{5.1}$$

where a, b, c, d are arbitrary scalar expressions. The *expression tree* for f looks like this:

TODO: Make figures.



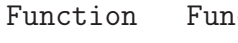
In **UFL** f is represented like this expression tree. If a,b,c,d are all distinct Function instances, the **UFL** representation will look like this:



If we instead have the expression

$$f = (a + b) * (a - b) \tag{5.2}$$

the tree will in fact look like this, with the functions a and b only represented once:



```
|         \       /       |
|          Sum         Product -- IntValue(-1)
|             |             |
|           Product    |
|             |             |
|---------- Sum ------|
```

The expression tree is a directed acyclic graph (DAG) where the vertices are Expr instances and each edge represents a direct dependency between two vertices, i.e. that one vertex is among the operands of another. A graph can also be represented in a linearized data structure, consisting of an array of vertices and an array of edges. This representation is convenient for many algorithms. An example to illustrate this graph representation:

```
G = V, E
V = [a, b, a+b, c, d, c+d, (a+b)*(c+d)]
E = [(6,2), (6,5), (5,3), (5,4), (2,0), (2,1)]
```

In the following this representation of an expression will be called the *computational graph*. To construct this graph from a **UFL** expression, simply do

```
G = Graph(expression)
V, E = G
```

The Graph class can build some useful data structures for use in algorithms.

```
Vin  = G.Vin()  # Vin[i]  = list of vertex indices j such that there is an edg
Vout = G.Vout() # Vout[i] = list of vertex indices j such that there is an edg
Ein  = G.Ein()  # Ein[i]  = list of edge indices j such that E[j] is an edge
Eout = G.Eout() # Eout[i] = list of edge indices j such that E[j] is an edge
```

The ordering of the vertices in the graph can in principle be arbitrary, but here they are ordered such that

$$v_i \prec v_j, \quad \forall j > i, \tag{5.3}$$

where $a \prec b$ means that a does not depend on b directly or indirectly.

Another property of the computational graph built by **UFL** is that no identical expression is assigned to more than one vertex. This is achieved efficiently by inserting expressions in a dict (a hash map) during graph building.

In principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each one separately. However, we can do better than that.

5.4.2 Partitioning the graph

To help generate better code efficiently, we can partition vertices by their dependencies, which allows us to, e.g., place expressions outside the quadrature loop if they don't depend (directly or indirectly) on the spatial coordinates. This is done simply by

```
P = partition(G) # TODO
```

TODO: finish dependencies.py

```
TODO
```

Bibliography

[1] M. Alnæs and K.-A. Mardal, *SyFi*, 2007. URL: http://www.fenics.org/syfi/.

[2] M. S. Alnæs and A. Logg, *UFL*, 2009. URL: http://www.fenics.org/ufl/.

[3] M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen, *UFC*, 2009. URL: http://www.fenics.org/ufc/.

[4] J. Hoffman, J. Jansson, C. Johnson, M. G. Knepley, R. C. Kirby, A. Logg, L. R. Scott, and G. N. Wells, *FEniCS*, 2006. URL: http://www.fenics.org/.

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

[6] A. Logg, *FFC*, 2007. URL: http://www.fenics.org/ffc/.

Appendix A

Commandline utilities

A.1 Validation and debugging: `ufl-analyse`

The command `ufl-analyse` loads all forms found in a `.ufl` file, tries to discover any errors in them, and prints various kinds of information about each form. Basic usage is

```
# ufl-analyse myform.ufl
```

For more information, type

```
# ufl-analyse --help
```

A.2 Formatting and visualization: `ufl-convert`

The command `ufl-convert` loads all forms found in a `.ufl` file, compiles them into a different form or extracts some information from them, and writes the result in a suitable file format.

To try this tool, go to the `demo/` directory of the **UFL** source tree. Some of the features to try are basic printing of `str` and `repr` string representations of each form:

```
# ufl-convert --format=str stiffness.ufl
# ufl-convert --format=repr stiffness.ufl
```

compilation of forms to mathematical notation in LaTeX:

```
# ufl-convert --filetype=pdf --format=tex --show=1 stiffness.ufl
```

LaTeX output of forms after processing with **UFL** compiler utilities:

```
# ufl-convert -tpdf -ftex -s1 --compile=1 stiffness.ufl
```

and visualization of expression trees using graphviz via compilation of forms to the dot format:

```
# ufl-convert -tpdf -fdot -s1 stiffness.ufl
```

Type `ufl-convert --help` for more details.

A.3 Conversion from FFC form files: `form2ufl`

The command `form2ufl` can be used to convert old FFC `.form` files to UFL format. To convert a form file named `myform.form` to UFL format, simply type

```
# form2ufl myform.ufl
```

Note that although, the `form2ufl` script may be helpful as a guide to converting old FFC `.form` files, it is not foolproof and may not always yield valid UFL files.

Appendix B

Installation

The source code of **UFL** 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 **UFL** mailing list at the address

```
ufl-dev@fenics.org
```

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

B.1 Installing from source

B.1.1 Dependencies and requirements

UFL currently has no external dependencies apart from a working Python installation.

Installing Python

UFL is developed for Python 2.5, and does not work with previous versions. 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`.

B.1.2 Downloading the source code

TODO: This section isn't yet correct, UFL hasn't been released officially yet.

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

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

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

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

This creates a directory `ufl-x.y.z` containing the **UFL** source code.

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

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

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 UFL

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

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

This will install the **UFL** Python package in a subdirectory called `ufl` in the default location for user-installed Python packages (usually something like `/usr/lib/python2.5/site-packages`).

In addition, the executable `ufl-analyse` (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 **UFL** in your home directory:

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

This installs the **UFL** package in the directory `~/local/lib/python` and the **UFL** executables 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 Running the test suite

To verify that the installation is correct, you may run the test suite. Enter the sub directory `test/` from within the **UFL** source tree and run the script `test.py`

```
# python test.py
```

This script runs all unit tests and imports **UFL** in the process.

B.2 Debian (Ubuntu) package

In preparation.

Appendix C

License

UFL is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

The GNU GPL is included verbatim below.

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
```

any other work released this way by its authors. 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 them 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 prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. 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.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to

avoid the special danger that patents applied to a free program could
make it effectively proprietary. To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

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

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License. Each licensee is addressed as "you". "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy. The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy. Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies. Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License. If

```
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.
```

```
  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey 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;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.
```

```
  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
```

```
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
```

```
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or
```

```
    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
```

violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License. If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program. Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance. However,
nothing other than this License grants you permission to propagate or
modify any covered work. These actions infringe copyright if you do
not accept this License. Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License. You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations. If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License. For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for

sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or

but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU 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 that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

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.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS

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.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

```
Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate
parts of the General Public License.  Of course, your program’s commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

Index