# Contents

1 Preliminaries 3  
1.1 Installation .................................................. 3  
1.2 Help and support ............................................. 3  
1.3 Development and reporting bugs ............................ 3  

2 Manual and API Reference 5  
2.1 UFL user manual ............................................... 5  
2.2 ufl package .................................................... 46  
2.3 Release notes .................................................. 185  

Python Module Index 187
This is the documentation for the Unified Form Language from the FEniCS Project (http://fenicsproject.org). The Unified Form Language (UFL) is a domain specific language for declaration of finite element discretizations of variational forms. More precisely, it defines a flexible interface for choosing finite element spaces and defining expressions for weak forms in a notation close to mathematical notation. UFL is described in the paper


UFL is most commonly used as the input language for the FEniCS Form Compiler (FFC) and in combination with the problem solving environment DOLFIN.
1.1 Installation

1.1.1 Ubuntu package

UFL may be installed directly from source, but the Debian (Ubuntu) package python-ufl is also available for UFL, as for other FEniCS components.

1.1.2 Manual from source

To retrieve the latest development version of UFL:

```
git clone https://bitbucket.org/fenics-project/ufl
```

To install UFL:

```
python setup.py install
```

1.2 Help and support

Send help requests and questions to fenics-support@googlegroups.com.
Send feature requests and questions to fenics-dev@googlegroups.com

1.3 Development and reporting bugs

The git source repository for UFL is located at https://bitbucket.org/fenics-project/ufl. For general UFL development questions and to make feature requests, use fenics-dev@googlegroups.com

Bugs can be registered at https://bitbucket.org/fenics-project/ufl/issues.
2.1 UFL user manual

2.1.1 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 is part of the FEniCS Project, and is usually used in combination with other components from this project to compute solutions to partial differential equations. The form compiler FFC uses UFL as its end-user interface, producing implementations of the UFC interface as output. See DOLFIN 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, you should read Form language, and also Example forms. These two sections 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 remaining 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.

Internal representation details describes 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.

Algorithms explains the many algorithms available to work with UFL expressions, mostly intended to aid developers of form compilers. The algorithms include 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.

2.1.2 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 the Python language, and any Python code is valid in the definition of a form. In particular, comments (lines starting with # and functions (keyword def, see user-defined 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.

### Forms and integrals

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

\[
a(v, w) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c(v, w) \, dx + \sum_{k=1}^{n_e} \int_{\partial \Omega_k} I_k^e(v, w) \, ds + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i(v, w) \, dS.
\]

Here the form \(a\) depends on the form arguments \(v = (v_1, \ldots, v_r)\) and the form coefficients \(w = (w_1, \ldots, w_n)\), 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 \(\Omega\) (\(dx\), cell integral);
- the boundary \(\partial \Omega\) of \(\Omega\) (\(ds\), exterior facet integral);
- the set of interior facets \(\Gamma\) (\(dS\), interior facet integral).

(Note that newer versions of UFL supports several other integral types currently not documented here). As a basic example, assume \(v\) is a scalar-valued expression and consider the integral of \(v\) over the interior of \(\Omega\). 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

\[
\partial \Omega_0 \subset \partial \Omega, \quad \partial \Omega_1 \subset \partial \Omega.
\]

**Note:** The domain \(\Omega\), its subdomains and boundaries are not known to UFL. These are defined in a problem solving environment such as DOLFIN, which uses UFL to specify forms.

### Finite element spaces

Before defining forms which can be integrated, it is necessary to describe the finite element spaces over which the integration takes place. UFL can represent very flexible general hierarchies of mixed finite elements, and has predefined names for most common element families. A finite element space is defined by an element domain, shape functions and nodal variables. In UFL, the element domain is called a **Cell**.
Cells

A polygonal cell is defined by a shape name and a geometric dimension, written as:

\[
\text{cell} = \text{Cell}(\text{shape, gdim})
\]

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

\[
\begin{align*}
\# & \text{ Regular triangle cell} \\
\text{cell} & = \text{Cell}(\text{"triangle"}) \\
\# & \text{ Triangle cell embedded in 3D space} \\
\text{cell} & = \text{Cell}(\text{"triangle", 3})
\end{align*}
\]

Objects for regular cells of all basic shapes are predefined:

\[
\begin{align*}
\# & \text{ Predefined linear cells} \\
\text{cell} & = \text{interval} \\
\text{cell} & = \text{triangle} \\
\text{cell} & = \text{tetrahedron} \\
\text{cell} & = \text{quadrilateral} \\
\text{cell} & = \text{hexahedron}
\end{align*}
\]

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.

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(div) elements;
- "Brezzi-Douglas-Fortin-Marini or "BDFM", representing vector-valued Brezzi–Douglas–Fortin–Marini H(div) elements;
- "Raviart-Thomas" or "RT", representing vector-valued Raviart–Thomas H(div) elements.
- "Nedelec 1st kind H(div)" or "N1div", representing vector-valued Nedelec H(div) elements (of the first kind).
- "Nedelec 2st kind H(div)" or "N2div", representing vector-valued Nedelec H(div) elements (of the second kind).
- "Nedelec 1st kind H(curl)" or "N1curl", representing vector-valued Nedelec H(curl) elements (of the first kind).
- "Nedelec 2st kind H(curl)" or "N2curl", representing vector-valued Nedelec H(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;

Note that new versions of UFL also support notation from the Periodic Table of Finite Elements, currently not documented here.

Basic elements

A FiniteElement, sometimes 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:

\[
element = \text{FiniteElement}(\text{family}, \text{cell}, \text{degree})
\]

Some examples:

\[
\begin{align*}
\text{element} &= \text{FiniteElement}("Lagrange", \text{interval}, 3) \\
\text{element} &= \text{FiniteElement}("DG", \text{tetrahedron}, 0) \\
\text{element} &= \text{FiniteElement}("BDM", \text{triangle}, 1)
\end{align*}
\]

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:

\[
element = \text{VectorElement}(\text{family}, \text{cell}, \text{degree}[, \text{size}])
\]

Some examples:

\[
\begin{align*}
\# A quadratic "P2" vector element on a triangle \\
\text{element} &= \text{VectorElement}("CG", \text{triangle}, 2) \\
\# A linear 3D vector element on a 1D interval \\
\text{element} &= \text{VectorElement}("CG", \text{interval}, 1, \text{size}=3) \\
\# A six-dimensional piecewise constant element on a tetrahedron \\
\text{element} &= \text{VectorElement}("DG", \text{tetrahedron}, 0, \text{size}=6)
\end{align*}
\]

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:

\[
element = \text{TensorElement}(\text{family}, \text{cell}, \text{degree}[, \text{shape}, \text{symmetry}])
\]

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

Examples:

\[
\begin{align*}
\text{element} &= \text{TensorElement}("CG", \text{cell}, 2) \\
\text{element} &= \text{TensorElement}("DG", \text{cell}, 0, \text{shape}=(6,6)) \\
\text{element} &= \text{TensorElement}("DG", \text{cell}, 0, \text{symmetry}=\text{True}) \\
\text{element} &= \text{TensorElement}("DG", \text{cell}, 0, \text{symmetry}={(0,0): (1,1)})
\end{align*}
\]
Mixed elements

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

General notation for an arbitrary number of subelements:

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

Shorthand notation for two subelements:

```
 element = element1 * element2
```

Note: The `*` operator is left-associative, such that:

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

represents \((e_1 * e_2) * e_3\), i.e. this is a mixed element with two sub-elements \((e_1 * e_2)\) and \(e_3\).

See Form arguments 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)
```

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
```

Form arguments

Form arguments are divided in two groups, arguments and coefficients. An Argument represents an arbitrary basis function in a given discrete finite element space, while a Coefficient represents a function in a discrete finite element space that will be provided by the user at a later stage. The number of Arguments that occur in a Form equals the “arity” of the form.
Basis functions

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

\[ v = \text{Argument(element)} \]

Note that more than one Argument 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 Arguments can be used to construct tuples of Arguments, as illustrated here for a mixed Taylor–Hood element:

\[ v, q = \text{Arguments(TH)} \]
\[ u, p = \text{Arguments(TH)} \]

For an Argument 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:

\[ vq = \text{Argument(TH)} \]
\[ v, q = \text{split(up)} \]

A shorthand for this is in place called Arguments:

\[ v, q = \text{Arguments(TH)} \]

For convenience, TestFunction and TrialFunction are special instances of Argument 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 Argument:

\[ v = \text{TestFunction(element)} \]
\[ u = \text{TrialFunction(element)} \]
\[ v, q = \text{TestFunctions(TH)} \]
\[ u, p = \text{TrialFunctions(TH)} \]

Meshes and function spaces

Note that newer versions of UFL introduce the concept of a Mesh and a FunctionSpace. These are currently not documented here.

Coefficient functions

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

\[ f = \text{Coefficient(element)} \]

Note that the order in which Coefficients are declared is important, directly reflected in the ordering they have among the arguments to each Form they are part of.

Coefficient is used to represent user-defined functions, including, e.g., source terms, body forces, variable coefficients and stabilization terms. UFL treats each Coefficient 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: Note that more than one function can be declared for the same FiniteElement. The following example declares two Arguments and two Coefficients for the same FiniteElement:

```python
v = Argument(element)
u = Argument(element)
f = Coefficient(element)
g = Coefficient(element)
```

For a Coefficient 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:

```python
up = Coefficient(TH)
u, p = split(up)
```

A shorthand for this is in place called Coefficients:

```python
u, p = Coefficient(TH)
```

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

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

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

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

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

**Basic Datatypes**

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

**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 SpatialCoordinate(cell):

```python
# Linear form for a load vector with a sin(y) coefficient
v = TestFunction(element)
x = SpatialCoordinate(cell)
L = sin(x[1])*v*dx
```

Another quantity is the (outwards pointing) facet normal FacetNormal(cell). 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:
\[ n = \text{FacetNormal}(\text{cell}) \]
\[ g = \text{Coefficient}(\text{VectorElement}("CG", \text{cell}, 1)) \]
\[ M = \text{dot}(n, g) \ast ds \]

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:

```python
# Geometric dimension
d = cell.geometric_dimension()

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

# Kronecker delta
delta_{ij} = I[i, j]
```

### 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`:

- \( v[i] \ast w[i] : \sum_{i=0}^{n-1} v[i] \cdot w[i] = v \cdot w \)
- \( \text{Dx}(v, i) \ast \text{Dx}(w, i) : \sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i} \frac{\partial w}{\partial x_i} = \nabla v \cdot \nabla w \)
- \( \text{Dx}(v[i], i) : \sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i} = \nabla v \)
- \( \text{Dx}(v[i], j) \ast \text{Dx}(w[i], j) : \sum_{i=0}^{n-1} \sum_{j=0}^{d-1} \frac{\partial v}{\partial x_i} \frac{\partial w}{\partial x_j} = \nabla v : \nabla w \)

Here we will 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 \( 1 \leq d \leq 3 \), and a set of orthonormal basis vectors \( i_i \) for \( i \in 0, \ldots, d-1 \), we can define the dot product of any two basis functions as

\[
    i_i \cdot i_j = \delta_{ij},
\]

where \( \delta_{ij} \) is the Kronecker delta

\[
    \delta_{ij} = \begin{cases} 
    1, & i = j, \\
    0, & \text{otherwise}.
    \end{cases}
\]

A rank 1 tensor (vector) quantity \( v \) 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:

\[
    v = v_k i_k \equiv \sum_k v_k i_k.
\]

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

\[
    A = A_{ij} i_i i_j \equiv \sum_i \sum_j A_{ij} i_i i_j.
\]

This generalizes to tensors of arbitrary rank:

\[
    C = C_{i_0} \otimes \cdots \otimes i_{r-1} \\
    \equiv \sum_{i_0} \cdots \sum_{i_{r-1}} C_{i_0} \otimes \cdots \otimes i_{r-1},
\]
where $C$ is a rank $r$ tensor and $\iota$ is a multi-index of length $r$.

When writing equations on paper, a mathematician can easily switch between the $v$ 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 $v$ 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 $(v, A)$ 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.

### 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)`:  

```python
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.

**Note:** Advanced usage

If using UFL inside DOLFIN 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:

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

**Note:** Advanced usage

Note that in the old FFC notation, the definition

```python
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:

```python
i = 0
```

### 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. \( \text{rank}(A[i,:]) == 1 \)
- \( A[:, j] \): column access, representing some column \( j \) of \( A \), i.e. \( \text{rank}(A[:,j]) == 1 \)
- \( C[\ldots, 0] \): subtensor access, representing the subtensor of \( A \) with the last axis fixed, e.g., \( A[\ldots,0] == A[:,0] \)
- \( C[j,\ldots] \): subtensor access, representing the subtensor of \( A \) with the last axis fixed, e.g., \( A[j,\ldots] == A[j,:] \)

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 \( v \) or \( A_{ij} \) to \( A \), the following examples show how this is done:

\[
v_k = \text{Identity}(\text{cell}.d)[0,k]
\]

\[
v = \text{as_tensor}(v_k, (k,))
\]

\[
A_{ij} = v[i] * u[j]
\]

\[
A = \text{as_tensor}(A_{ij}, (i,j))
\]

Here \( v \) will represent unit vector \( i_0 \), 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:

\[
v = \text{as_vector}([1.0, 2.0, 3.0])
\]

\[
A = \text{as_matrix}([[u[0], 0], [0, u[1]])
\]

\[
B = \text{as_matrix}([[a+b for b in range(2)] for a in range(2)])
\]

Here \( v, A \) and \( B \) will represent the expressions

\[
v = i_0 + 2i_1 + 3i_2,
\]

\[
A = \begin{bmatrix} u_0 & 0 \\ 0 & u_1 \end{bmatrix},
\]

\[
B = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}.
\]

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

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 sometimes written $v_{i,i}$ and $A_{ij,i}$ in pen-and-paper index notation.

**Basic algebraic operators**

The basic algebraic operators $+, -, \ast, /$ 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, $a \ast b$:**
- 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.

**Basic nonlinear functions**

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

- $\text{abs}(f)$: the absolute value of $f$.
- $\text{sign}(f)$: the sign of $f$ (+1 or -1).
- $\text{pow}(f, g)$ or $f^{**}g$: $f$ to the power $g$, $f^g$
- $\text{sqrt}(f)$: square root, $\sqrt{f}$
- $\text{exp}(f)$: exponential of $f$
- $\text{ln}(f)$: natural logarithm of $f$
- $\text{cos}(f)$: cosine of $f$
- $\text{sin}(f)$: sine of $f$
- $\text{tan}(f)$: tangent of $f$
- $\text{cosh}(f)$: hyperbolic cosine of $f$
- $\text{sinh}(f)$: hyperbolic sine of $f$
• \( \tanh(f) \): hyperbolic tangent of \( f \)
• \( \acos(f) \): inverse cosine of \( f \)
• \( \asin(f) \): inverse sine of \( f \)
• \( \atan(f) \): inverse tangent of \( f \)
• \( \atan2(f1, f2) \): inverse tangent of \( \frac{f1}{f2} \)
• \( \erf(f) \): error function of \( f \),
\[
\frac{2}{\sqrt{\pi}} \int_0^f \exp(-t^2) \, dt
\]
• \( \bessel{J}(\nu, f) \): Bessel function of the first kind, \( J_\nu(f) \)
• \( \bessel{Y}(\nu, f) \): Bessel function of the second kind, \( Y_\nu(f) \)
• \( \bessel{I}(\nu, f) \): Modified Bessel function of the first kind, \( I_\nu(f) \)
• \( \bessel{K}(\nu, f) \): Modified Bessel function of the second kind, \( K_\nu(f) \)

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

**Tensor algebra operators**

**transpose**

The transpose of a matrix \( A \) can be written as:

\[
\begin{align*}
AT &= \text{transpose}(A) \\
AT &= A^T \\
AT &= \text{as}\_\text{matrix}(A[i,j], (j,i))
\end{align*}
\]

The definition of the transpose is

\[
AT[i,j] \leftrightarrow (A^T)_{ij} = A_{ji}
\]

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

\[
AT = \text{as}\_\text{tensor}(A[i,j,k,l], (l,k,j,i))
\]

**tr**

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

\[
\begin{align*}
t &= \text{tr}(A) \\
t &= A[i,i]
\end{align*}
\]

The definition of the trace is

\[
\text{tr}(A) \leftrightarrow \text{tr}A = A_{ii} = \sum_{i=0}^{n-1} A_{ii}.
\]

**dot**

The dot product of two tensors \( a \) and \( b \) can be written:
The definition of the dot product of unit vectors is (assuming an orthonormal basis for a Euclidean space):

\[ \mathbf{i}_i \cdot \mathbf{i}_j = \delta_{ij} \]

where \( \delta_{ij} \) is the Kronecker delta function. The dot product of higher order tensors follow from this, as illustrated with the following examples.

An example with two vectors

\[ \mathbf{v} \cdot \mathbf{u} = (v_i \mathbf{i}_i) \cdot (u_j \mathbf{i}_j) = v_i u_j (\mathbf{i}_i \cdot \mathbf{i}_j) = v_i u_j \delta_{ij} = v_i u_i \]

An example with a tensor of rank two

\[ \mathbf{A} \cdot \mathbf{B} = (A_{ij} \mathbf{i}_i \mathbf{i}_j) \cdot (B_{kl} \mathbf{i}_k \mathbf{i}_l) \]
\[ = (A_{ij} B_{kl}) \mathbf{i}_i (\mathbf{i}_j \cdot \mathbf{i}_k) \mathbf{i}_l \]
\[ = (A_{ij} B_{kl} \delta_{jk}) \mathbf{i}_i \mathbf{i}_l \]
\[ = A_{ik} B_{kl} \mathbf{i}_i \mathbf{i}_l. \]

This is the same as a matrix-matrix multiplication.

An example with a vector and a tensor of rank two

\[ \mathbf{v} \cdot \mathbf{A} = (v_j \mathbf{i}_j) \cdot (A_{kl} \mathbf{i}_k \mathbf{i}_l) \]
\[ = (v_j A_{kl}) (\mathbf{i}_j \cdot \mathbf{i}_k) \mathbf{i}_l \]
\[ = (v_j A_{kl} \delta_{jk}) \mathbf{i}_l \]
\[ = v_k A_{kl} \mathbf{i}_l. \]

This is the same as a 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 \( \text{rank}(a)+\text{rank}(b)-2 \).

### inner

The inner product is a contraction over all axes of \( a \) and \( b \), that is the sum of all component-wise products. The operands must have exactly the same dimensions. For two vectors it is equivalent to the dot product.

If \( \mathbf{A} \) and \( \mathbf{B} \) are rank two tensors and \( \mathbf{C} \) and \( \mathbf{D} \) are rank 3 tensors their inner products are

\[ \mathbf{A} : \mathbf{B} = A_{ij} B_{ij} \]
\[ \mathbf{C} : \mathbf{D} = C_{ijk} D_{ijk} \]

Using UFL notation, the following sets of declarations are equivalent:

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

\[
f = \text{inner}(A, B)
\]
\[
f = A[i,j] \times B[i,j]
\]

# Rank 3 tensors

\[
f = \text{inner}(C, D)
\]
\[
f = C[i,j,k] \times D[i,j,k]
\]

**outer**

The outer product of two tensors \(a\) and \(b\) can be written:

\[
A = \text{outer}(a, b)
\]

The general definition of the outer product of two tensors \(C\) of rank \(r\) and \(D\) of rank \(s\) is

\[
C \otimes D = C_{i_0 \ldots i_{r-1}} D_{k_0 \ldots k_{s-1}} i_0 \otimes \cdots \otimes i_{r-2} \otimes i_1 \otimes \cdots \otimes i_{s-1}
\]

Some examples with vectors and matrices are easier to understand:

\[
v \otimes u = v_i u_j i_{ij},
\]
\[
v \otimes v = v_i B_{kl} i_k i_l,
\]
\[
A \otimes B = A_{ij} B_{kl} i_j i_k i_l.
\]

The outer product of vectors is often written simply as:

\[
v \otimes u = vu,
\]

which is what we have done with \(i_i i_j\) above.

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

**cross**

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

\[
\text{cross}(v, w) \leftrightarrow v \times w = (v_1 w_2 - v_2 w_1, v_2 w_0 - v_0 w_2, v_0 w_1 - v_1 w_0)
\]

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

**det**

The determinant of a matrix \(A\) can be written:

\[
d = \text{det}(A)
\]

**dev**

The deviatoric part of matrix \(A\) can be written:
\[ B = \text{dev}(A) \]

**sym**

The symmetric part of \( A \) can be written:

\[ B = \text{sym}(A) \]

The definition is

\[ \text{sym}A = \frac{1}{2}(A + A^T) \]

**skew**

The skew symmetric part of \( A \) can be written:

\[ B = \text{skew}(A) \]

The definition is

\[ \text{skew}A = \frac{1}{2}(A - A^T) \]

**cofac**

The cofactor of a matrix \( A \) can be written:

\[ B = \text{cofac}(A) \]

The definition is

\[ \text{cofac}A = \text{det}(A)A^{-1} \]

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.

**inv**

The inverse of matrix \( A \) can be written:

\[ A^{-1} = \text{inv}(A) \]

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.

**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.
Basic spatial derivatives

Spatial derivatives hold a special physical meaning in partial differential equations and there are several ways to express those. The basic way is:

```python
# 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 \( v \) with respect to spatial direction \( z \). If \( v \) has no free indices, \( g \) is the scalar derivative in spatial direction \( x_i \), and \( g \) has the free index \( i \). This can be expressed compactly as \( v, i \):

\[
f = \frac{\partial v}{\partial x_2} = v_{,2},
g = \frac{\partial v}{\partial x_i} = v_{,i}.
\]

If the expression to be differentiated w.r.t. \( x_i \) has \( i \) as a free-index, implicit summation is implied:

```python
# 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

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

Note: \( v[i].dx(i) \) and \( v_{,i} \) with compact notation denote implicit summation.

Compound spatial derivatives

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

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

The operand \( f \) can have no free indices.

Gradient

The gradient of a scalar \( u \) is defined as

\[
\text{grad}(u) \equiv \nabla u = \sum_{k=0}^{d-1} \frac{\partial u}{\partial x_k} \mathbf{i}_k,
\]
which is a vector of all spatial partial derivatives of \( u \).

The gradient of a vector \( v \) is defined as

\[
\text{grad}(v) \equiv \nabla v = \frac{\partial v_i}{\partial x_j} i_i j,
\]

which written componentwise is

\[
A = \nabla v, \quad A_{ij} = v_{i,j}.
\]

In general for a tensor \( A \) of rank \( r \) the definition is

\[
\text{grad}(A) \equiv \nabla A = (\frac{\partial}{\partial x^i})(A_{i_0} \otimes \cdots \otimes i_{r-1}) \otimes i_i = \frac{\partial A_{i_0} \otimes \cdots \otimes i_{r-1}}{\partial x^i} \otimes i_i,
\]

where \( i \) is a multiindex of length \( r \).

In UFL, the following pairs of declarations are equivalent:

\[
\begin{align*}
D f i &= \text{grad}(f)[i] \\
D f i &= f.dx(i) \\
D v i &= \text{grad}(v)[i, j] \\
D v i &= v[i].dx(j) \\
D A i &= \text{grad}(A)[..., i] \\
D A i &= A[..., i].dx(i)
\end{align*}
\]

for a scalar expression \( f \), a vector expression \( v \), and a tensor expression \( A \) of arbitrary rank.

**Divergence**

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

The divergence of a vector \( v \) is defined as

\[
\text{div}(v) \equiv \nabla \cdot v = \sum_{k=0}^{d-1} \frac{\partial v_k}{\partial x_k}
\]

In UFL, the following declarations are equivalent:

\[
\begin{align*}
d v &= \text{div}(v) \\
d v &= v[i].dx(i) \\
d A &= \text{div}(A) \\
d A &= A[... , i].dx(i)
\end{align*}
\]

for a vector expression \( v \) and a tensor expression \( A \).

**Curl and rot**

The operator \texttt{curl} or \texttt{rot} accepts as argument a vector-valued expression and returns its curl:

\[
\text{curl}(v) = \nabla \times v = \left( \frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1} \right).
\]
Note: The \textit{curl} or \textit{rot} operator is only defined for vectors of length three.

In UFL, the following declarations are equivalent:

\begin{verbatim}
omega = curl(v)
omega = rot(v)
\end{verbatim}

Variable derivatives

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

The notation is illustrated here:

\begin{verbatim}
# Define some arbitrary expression
u = Coefficient(element)
w = sin(u**2)

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

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

# The derivative of expression f w.r.t. the variable w
df = diff(f, w)
\end{verbatim}

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 \( t \):

\begin{verbatim}
t = variable(Constant(cell))
f = sin(x[0])**2 * cos(t)
dfdt = diff(f, t)
\end{verbatim}

Functional derivatives

The third and final kind of derivative are derivatives of functionals or forms w.r.t. a \texttt{Coefficient}. This is described in more detail in the section \textit{AD} about form transformations.

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.

\textbf{Restriction}: \( v(\cdot^+ \cdot) \) and \( v(\cdot^- \cdot) \)

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 = Coefficient(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.

Jump: \texttt{jump(v)}

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

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

\[
\text{jump}(v) \leftrightarrow \begin{bmatrix} v \end{bmatrix} = v^+ - v^-
\]

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

If called with two arguments, \texttt{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 \texttt{jump(v, n)} is given by

\[
\text{jump}(v, n) \leftrightarrow \begin{bmatrix} v \end{bmatrix} = v^+ n^+ + v^- n^-
\]

If \(v\) is vector-valued, then \texttt{jump(v, n)} is given by

\[
\text{jump}(v, n) \leftrightarrow \begin{bmatrix} v \end{bmatrix} = v^+ \cdot n^+ + v^- \cdot n^-
\]

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

Average: \texttt{avg(v)}

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

\[
\text{avg}(v) \leftrightarrow \begin{bmatrix} v \end{bmatrix} = \frac{1}{2}(v^+ + v^-)
\]

The expression \texttt{avg(v)} has the same value shape as the expression \(v\).

Conditional Operators

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).

**Conditions**

- `eq(a, b)` must be used in place of the notation `a == b`
- `ne(a, b)` must be used in place of the notation `a != b`
- `le(a, b)` is equivalent to `a <= b`
- `ge(a, b)` is equivalent to `a >= b`
- `lt(a, b)` is equivalent to `a < b`
- `gt(a, b)` is equivalent to `a > b`

**Note:** Because of details in the way Python behaves, we cannot overload the `==` operator hence these named operators.

**User-defined operators**

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

$$\varepsilon(v) = \frac{1}{2}(\nabla v + (\nabla v)^T).$$

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

```python
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:

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

**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.

**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 `Coefficient` with a fixed value for optimized runtime evaluation.

Example:
\[ f = \text{Coefficient}(\text{element}) \]
\[ g = \text{Coefficient}(\text{element}) \]
\[ c = \text{Constant}(\text{cell}) \]
\[ a = f \cdot g \cdot v \cdot dx \]
\[ b = \text{replace}(a, \{ f: 3.14, g: c \}) \]

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

**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:

\[
L = \text{action}(a, w) \\
f = \text{action}(L, w)
\]

To give a concrete example, these declarations are equivalent:

\[
\begin{align*}
a &= \text{inner}(\text{grad}(u), \text{grad}(v)) \cdot dx \\
L &= \text{action}(a, w) \\
\end{align*}
\]

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.)

**Energy norm of a bilinear form**

The functional representing the energy norm \( |v|_A = v^T Av \) of a matrix \( A \) assembled from a form \( a \) can be computed with:

\[ f = \text{energy\_norm}(a, w) \]

which is equivalent to:

\[ f = \text{action}(\text{action}(a, w), w) \]
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:

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

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 assembly, 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`:

```python
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:

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

As a slightly more complicated example, this formulation:

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

is equivalent to:

```python
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
```

Automatic functional differentiation

UFL can compute derivatives of functionals or forms w.r.t. to a `Coefficient`. 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 can be differentiated to obtain the bilinear form corresponding to its Jacobi matrix.

Note: 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.
\[ J(v, u; w) = \frac{d}{dw} F(v; w). \]

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

\[
\begin{align*}
f &= (w^2)/2 \times dx \\
F &= \text{derivative}(f, w, v) \\
J &= \text{derivative}(F, w, u)
\end{align*}
\]

which is equivalent to:

\[
\begin{align*}
f &= (w^2)/2 \times dx \\
F &= w \times v \times dx \\
J &= u \times v \times dx
\end{align*}
\]

Assume in the following examples that:

\[
\begin{align*}
v &= \text{TestFunction}(\text{element}) \\
u &= \text{TrialFunction}(\text{element}) \\
w &= \text{Coefficient}(\text{element})
\end{align*}
\]

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

\[
\begin{align*}
f &= \text{inner}(\nabla(w), \nabla(w))/2 \times dx \\
F &= \text{derivative}(f, w, v) \\
J &= \text{derivative}(F, w, u)
\end{align*}
\]

which is equivalent to:

\[
\begin{align*}
f &= \text{inner}(\nabla(w), \nabla(w))/2 \times dx \\
F &= \text{inner}(\nabla(w), \nabla(v)) \times dx \\
J &= \text{inner}(\nabla(u), \nabla(v)) \times dx
\end{align*}
\]

Note that here the basis functions are provided explicitly, which is some times necessary, e.g., if part of the form is linearized manually as in:

\[
\begin{align*}
g &= \text{Coefficient}(\text{element}) \\
f &= \text{inner}(\nabla(w), \nabla(w)) \times dx \\
F &= \text{derivative}(f, w, v) + \text{dot}(w-g,v) \times dx \\
J &= \text{derivative}(F, w, u)
\end{align*}
\]

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:

\[
\begin{align*}
X &= \text{VectorElement}("\text{Lagrange}", \text{cell}, 1) \\
Y &= \text{FiniteElement}("\text{Lagrange}", \text{cell}, 1) \\
x &= \text{Coefficient}(X) \\
y &= \text{Coefficient}(Y) \\
L &= \text{inner}(\nabla(x), \nabla(x)) \times dx + \text{dot}(x,x) \times y \times dx \\
F &= \text{derivative}(L, (x,y)) \\
J &= \text{derivative}(F, (x,y))
\end{align*}
\]

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

\[
\begin{align*}
    u &= \text{Coefficient}(M) \\
    x, y &= \text{split}(u) \\
    L &= \text{inner}(\text{grad}(x), \text{grad}(x)) \cdot dx + \text{dot}(x, x) \cdot y \cdot dx \\
    F &= \text{derivative}(L, u) \\
    J &= \text{derivative}(F, u)
\end{align*}
\]

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 **Coefficient** represents a sum of unknown coefficients multiplied with unknown basis functions in some finite element space.

\[
w(x) = \sum_k w_k \phi_k(x)
\]

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

\[
v(x) = \phi_k(x), \quad \phi_k \in V_h.
\]

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

\[
b_i = L(\phi_i; \sum_k w_k \phi_k), \quad \forall \phi_i \in V_h.
\]

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

\[
A_{ij} = \frac{d b_i}{d w_j} = a(\phi_i, \phi_j; \sum_k w_k \phi_k), \quad \forall \phi_i \in V_h, \quad \forall \phi_j \in V_h,
\]

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 \( \Omega \) is independent of \( w_j \), \( \int_\Omega \) commutes with \( \frac{d}{d w_j} \), and we can differentiate the integrand expression instead, e.g.,

\[
L(v; w) = \int_\Omega L_c(v; w) \, dx + \int_{\partial\Omega} L_e(v; w) \, ds,
\]

\[
\frac{d}{d w_j} L(v; w) = \int_\Omega \frac{d L_c}{d w_j} \, dx + \int_{\partial\Omega} \frac{d L_e}{d w_j} \, ds.
\]

In addition, we need that

\[
\frac{d w}{d w_j} = \phi_j, \quad \forall \phi_j \in V_h,
\]

which in UFL can be represented as

\[
w = \text{Coefficient}(\text{element}), \\
v = \text{Argument}(\text{element}), \\
\frac{d w}{d w_j} = v,
\]

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.

28 Chapter 2. Manual and API Reference
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:

```python
element = FiniteElement("CG", cell, 1)
w = Coefficient(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 = Coefficient(element)
Jnorm = energy_norm(J, g)
```

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:

```python
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.

### 2.1.3 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.

#### The mass matrix

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

\[ a(v, u) = \int_{\Omega} v u \, dx, \]

which can be implemented in UFL as follows:

```python
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.

### Poisson equation

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

\[
a(v, u) = \int_{\Omega} \nabla v \cdot \nabla u \, dx,
\]

\[
L(v; f) = \int_{\Omega} v \, f \, dx,
\]

can be implemented as follows:

```python
element = FiniteElement("Lagrange", triangle, 1)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(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:

\[
a = \text{Dx}(v, i) \cdot \text{Dx}(u, i) \cdot dx
\]

or like this:

\[
a = v.dx(i) \cdot u.dx(i) \cdot dx
\]

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

### Vector-valued Poisson

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

\[
a(v, u) = \int_{\Omega} \nabla v : \nabla u \, dx,
\]

\[
L(v; f) = \int_{\Omega} v \cdot f \, dx,
\]

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

```python
element = VectorElement("Lagrange", triangle, 1)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)
a = inner(grad(v), grad(u))*dx
L = dot(v, f)*dx
```

Alternatively, index notation may be used like this:
\[ a = D_x(v[i], j) \cdot D_x(u[i], j) \cdot dx \]
\[ L = v[i] \cdot f[i] \cdot dx \]

or like this:
\[ a = v[i].dx(j) \cdot u[i].dx(j) \cdot dx \]
\[ L = v[i] \cdot f[i] \cdot dx \]

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

**The strain-strain term of linear elasticity**

The strain-strain term of linear elasticity,
\[ a(v, u) = \int_{\Omega} \epsilon(v) : \epsilon(u) \, dx, \]
where
\[ \epsilon(v) = \frac{1}{2} (\nabla v + (\nabla v)^T) \]
can be implemented as follows:

```python
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:
\[ a = 0.25 \cdot (D_x(v[j], i) + D_x(v[i], j)) \cdot (D_x(u[j], i) + D_x(u[i], j)) \cdot dx \]

or like this:
\[ a = 0.25 \cdot (v[j].dx(i) + v[i].dx(j)) \cdot (u[j].dx(i) + u[i].dx(j)) \cdot dx \]

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

**The nonlinear term of Navier–Stokes**

The bilinear form for fixed-point iteration on the nonlinear term of the incompressible Navier–Stokes equations,
\[ a(v, u; w) = \int_{\Omega} (w \cdot \nabla u) \cdot v \, dx, \]
with \( w \) the frozen velocity from a previous iteration, can be implemented as follows:
**The heat equation**

Discretizing the heat equation,

\[ \dot{u} - \nabla \cdot (c \nabla u) = f, \]

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 \( u^n_h = u_h(\cdot, t_n) \) with \( u_h^{n-1} = u_h(\cdot, t_{n-1}) \) given such that

\[
\frac{1}{k_n} \int_{\Omega} u^n_h - u^{n-1}_h \, dx + \int_{\Omega} c \nabla v \cdot \nabla u^n_h \, dx = \int_{\Omega} v \, f^n \, dx
\]

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 \( a(v, u_h^n) = L(v) \) for all \( v \), we obtain the following pair of bilinear and linear forms:

\[
a(v, u_h^n; c, k) = \int_{\Omega} v u_h^n \, dx + k_n \int_{\Omega} c \nabla v \cdot \nabla u_h^n \, dx,
\]

\[
L(v; u_h^{n-1}, f, k) = \int_{\Omega} v u_h^{n-1} \, dx + k_n \int_{\Omega} v \, f^n \, dx,
\]

which can be implemented as follows:

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

v = TestFunction(element)  # Test function
u1 = TrialFunction(element)  # Value at t_n
u0 = Coefficient(element)  # Value at t_{n-1}
c = Coefficient(element)  # Heat conductivity
f = Coefficient(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.
**Mixed formulation of Stokes**

To solve Stokes’ equations,

\[-\Delta u + \nabla p = f, \]
\[\nabla \cdot u = 0,\]

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:

\[a((v, q), (u, p)) = \int_{\Omega} \nabla v \cdot \nabla u - (\nabla \cdot v) p + q (\nabla \cdot u) \, dx,\]
\[L((v, q); f) = \int_{\Omega} v \cdot f \, dx.\]

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

```python
cell = triangle
P2 = VectorElement("Lagrange", cell, 2)
P1 = FiniteElement("Lagrange", cell, 1)
TH = P2 * P1
(v, q) = TestFunctions(TH)
(u, p) = TrialFunctions(TH)
f = Coefficient(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.

**Mixed formulation of Poisson**

We next consider the following formulation of Poisson’s equation as a pair of first order equations for \(\sigma \in H(\text{div})\) and \(u \in L^2\):

\[\sigma + \nabla u = 0,\]
\[\nabla \cdot \sigma = f.\]

We multiply the two equations by a pair of test functions \(\tau \) and \(w\) and integrate by parts to obtain the following variational problem: Find \((\sigma, u) \in V = H(\text{div})\times L^2\) such that

\[a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall (\tau, w) \in V,\]

where

\[a((\tau, w), (\sigma, u)) = \int_{\Omega} \tau \cdot \sigma - \nabla \cdot \tau u + w \nabla \cdot \sigma \, dx,\]
\[L((\tau, w); f) = \int_{\Omega} w \cdot f \, dx.\]

We may implement the corresponding forms in our form language using first order BDM \(H(\text{div})\)-conforming elements for \(\sigma\) 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 = Coefficient(DG0)

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

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

Poisson 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

\[
a(v, u; h) = \int_\Omega \nabla v \cdot \nabla u \, dx \\
+ \sum_S \int_S -(\nabla v) \cdot ([u])_n - ([v])_n \cdot (\nabla u) + (\alpha/h)[[v]]_n \cdot [[u]]_n \, dS \\
+ \int_\Gamma -\nabla v \cdot ([u])_n - ([v])_n \cdot \nabla u + (\gamma/h)v u \, ds
\]

\[
L(v; f, g) = \int_\Omega v f \, dx + \int_\partial\Omega v g \, ds.
\]

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 = Coefficient(DG1)
g = Coefficient(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
This example is implemented in the file `PoissonDG.ufl` in the collection of demonstration forms included with the UFL source distribution.

**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:

The linearised bilinear and linear forms for this equation, can be implemented in a single form file as follows:

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

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

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 three forms using the standard `FiniteElement` (linear elements) can then be implemented as:

```python
element = FiniteElement("Lagrange", triangle, 1)
DG = FiniteElement("Discontinuous Lagrange", triangle, 0)
sig = VectorElement("Discontinuous Lagrange", triangle, 0)

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

and:

```python
2.1. UFL user manual 35
```
element = FiniteElement("Lagrange", triangle, 1)
DG = FiniteElement("Discontinuous Lagrange", triangle, 0)

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

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

and:

element = FiniteElement("Lagrange", triangle, 1)
DG = VectorElement("Discontinuous Lagrange", triangle, 0)

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

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

The three 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)

and:

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

and:

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

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

Typical values of the relative residual for each Newton iteration for all three approaches are shown in Table~ref{tab:convergence1}. It is noted that the convergence rate is quadratic as it should be for all 3 methods.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>REF1</th>
<th>FE1</th>
<th>QE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.3e-02</td>
<td>6.3e-02</td>
<td>6.3e-02</td>
</tr>
<tr>
<td>2</td>
<td>5.3e-04</td>
<td>5.3e-04</td>
<td>5.3e-04</td>
</tr>
<tr>
<td>3</td>
<td>3.7e-08</td>
<td>3.7e-08</td>
<td>3.7e-08</td>
</tr>
<tr>
<td>4</td>
<td>2.9e-16</td>
<td>2.9e-16</td>
<td>2.5e-16</td>
</tr>
</tbody>
</table>

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 (tab:convergence2). Clearly, using the
standard \texttt{FiniteElement} leads to a poor convergence whereas the \texttt{QuadratureElement} still leads to quadratic convergence.

Relative residuals for each approach for quadratic elements:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>REF2</th>
<th>FE2</th>
<th>QE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.6e-01</td>
<td>3.9e-01</td>
<td>2.6e-01</td>
</tr>
<tr>
<td>2</td>
<td>1.1e-02</td>
<td>4.6e-02</td>
<td>1.1e-02</td>
</tr>
<tr>
<td>3</td>
<td>1.2e-05</td>
<td>1.1e-02</td>
<td>1.6e-05</td>
</tr>
<tr>
<td>4</td>
<td>1.1e-11</td>
<td>7.2e-04</td>
<td>9.1e-09</td>
</tr>
</tbody>
</table>

\textbf{More examples}

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

\%TODO: Modify rest of FFC example forms to UFL syntax and add here.

\textbf{2.1.4 Internal representation details}

FIXME: This chapter is very much outdated. Most of the concepts are still the same but a lot of the details are different.

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

\begin{verbatim}
from ufl.classes import *
\end{verbatim}

\textbf{Structure of a form}

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

Subclasses of \texttt{Terminal} represent atomic quantities which terminate the expression tree, e.g. they have no subexpressions. Subclasses of \texttt{Operator} represent operations on one or more other expressions, which may usually be \texttt{Expr} subclasses of arbitrary type. Different \texttt{Operators} 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.

\textbf{General properties of expressions}

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

\texttt{operands}

\texttt{u.operands()} returns a tuple with all the operands of \texttt{u}, which should all be \texttt{Expr} instances.
reconstruct

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

cell

\texttt{u.cell()} returns the first \texttt{Cell} instance found in \texttt{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 \texttt{None} and \texttt{u} is spatially constant. Note that this property is used in some algorithms.

shape

\texttt{u.shape()} returns a tuple of integers, which is the tensor shape of \texttt{u}.

free_indices

\texttt{u.free_indices()} returns a tuple of \texttt{Index} objects, which are the unassigned, free indices of \texttt{u}.

index_dimensions

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

str(u)

\texttt{str(u)} returns a human-readable string representation of \texttt{u}.

repr(u)

\texttt{repr(u)} returns a Python string representation of \texttt{u}, such that \texttt{eval(repr(u)) == u} holds in Python.

hash(u)

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

\texttt{u == v}

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

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.

Terminals

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

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.

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 `dolfin` type cannot be part of UFL. So before adding a new class, consider that doing so may require changes in multiple algorithms and even other projects.

2.1.5 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:

```python
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.

**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:

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

```python
str
```

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

```python
{ v_0 * v_1 * w_0 } * dx(<Mesh #-1 with coordinates parameterized by <Lagrange vector element of degree 1 on a triangle: 2 x <CG1 on a triangle>>)
+ { v_0 * w_1 } * ds(<Mesh #-1 with coordinates parameterized by <Lagrange vector element of degree 1 on a triangle: 2 x <CG1 on a triangle>>)
```

```python
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)`:

```python
Form([Integral(Product(Argument(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0, None) * Product(Argument(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 1, None) * Coefficient(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0)), 'exterior_facet', Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), 'everywhere', {}, None)])
```

**Tree formatting**

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

```python
Form:
  Integral:
    integral type: cell
    subdomain id: everywhere
    integrand:
      Product
        (Argument(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0, None) * Product(Argument(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 1, None) * Coefficient(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0))
  Integral:
    integral type: exterior_facet
    subdomain id: everywhere
    integrand:
```

Inspecting and manipulating the expression tree

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

Traversing expressions

```
iter_expressions  Example usage:
for e in iter_expressions(a):
    print str(e)
```

outputs:

```
v_0 * v_1 * w_0
v_0 * w_1
```

```
post_traversal  TODO: traversal.py
```

```
pre_traversal  TODO: traversal.py
```

```
walk  TODO: traversal.py
```

```
traverse_terminals  TODO: traversal.py
```

Extracting information

TODO: analysis.py

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 transformation 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):
```

2.1. UFL user manual
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:

```
Visiting v_0 * v_1 with operands:
v_0, v_1
```

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. `vector_constant` for `VectorConstant`. 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:

```python
from ufl.classes import *
class CoefficientReplacer(Transformer):
    def __init__(self):
        Transformer.__init__(self)

        expr = Transformer.reuse_if_possible
        terminal = Transformer.always_reuse

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

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

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

outputs:

```
3.14 * v_0
```

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_possible` 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 sometimes 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:

```python
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 = Coefficient(element)
g = Coefficient(element)
h = Coefficient(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.

### Automatic differentiation implementation

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

TODO: More details about AD algorithms for developers.

#### Forward mode

TODO: forward_ad.py

#### Reverse mode

TODO: reverse_ad.py

#### Mixed derivatives

TODO: ad.py
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 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.

% TODO: Finish and test this before writing about it :) %The vertices of a graph can be reordered to improve the efficiency of the generated code, an operation usually called operation scheduling.

The computational graph

TODO: finish graph.py:

TODO

Consider the expression:

\[
f = (a + b) \ast (c + d)
\]

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

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

If we instead have the expression

\[
f = (a + b) \ast (a - b)
\]

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

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:

\[ G = \text{Graph}(\text{expression}) \]
\[ V, E = G \]

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

\[ \text{Vin} = G.\text{Vin}() \quad \# \text{Vin}[i] = \text{list of vertex indices } j \text{ such that there is an edge from } V[j] \text{ to } V[i] \]
\[ \text{Vout} = G.\text{Vout}() \quad \# \text{Vout}[i] = \text{list of vertex indices } j \text{ such that there is an edge from } V[i] \text{ to } V[j] \]
\[ \text{Ein} = G.\text{Ein}() \quad \# \text{Ein}[i] = \text{list of edge indices } j \text{ such that } E[j] \text{ is an edge to } V[i], \text{ e.g. } E[j][1] = i \]
\[ \text{Eout} = G.\text{Eout}() \quad \# \text{Eout}[i] = \text{list of edge indices } j \text{ such that } E[j] \text{ is an edge from } V[i], \text{ e.g. } E[j][0] = i \]

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

\[ v_i < v_j, \quad \forall j > i, \]

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.

**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:

\[ P = \text{partition}(G) \quad \# \text{TODO} \]

### 2.1.6 Commandline utilities

**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

**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:

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

Compilation of forms to mathematical notation in LaTeX:

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

LaTeX output of forms after processing with UFL compiler utilities:

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

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

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

Type ufl-convert --help for more details.

# 2.2  ufl package

## 2.2.1 Subpackages

### ufl.algorithms package

**Submodules**

#### ufl.algorithms.ad module

Front-end for AD routines.

```python
ufl.algorithms.ad.expand_derivatives(form, **kwargs)
```

Expand all derivatives of expr.

In the returned expression g which is mathematically equivalent to expr, there are no VariableDerivative or CoefficientDerivative objects left, and Grad objects have been propagated to Terminal nodes.

#### ufl.algorithms.analysis module

Utility algorithms for inspection of and information extraction from UFL objects in various ways.

```python
ufl.algorithms.analysis.extract_arguments(a)
```

Build a sorted list of all arguments in a, which can be a Form, Integral or Expr.

```python
ufl.algorithms.analysis.extract_arguments_and_coefficients(a)
```

Build two sorted lists of all arguments and coefficients in a, which can be a Form, Integral or Expr.

```python
ufl.algorithms.analysis.extract_coefficients(a)
```

Build a sorted list of all coefficients in a, which can be a Form, Integral or Expr.

```python
ufl.algorithms.analysis.extract_elements(form)
```

Build sorted tuple of all elements used in form.

```python
ufl.algorithms.analysis.extract_sub_elements(elements)
```

Build sorted tuple of all sub elements (including parent element).
ufl.algorithms.analysis.extract_type(a, ufl_type)
    Build a set of all objects of class ufl_type found in a. The argument a can be a Form, Integral or Expr.

ufl.algorithms.analysis.extract_unique_elements(form)
    Build sorted tuple of all unique elements used in form.

ufl.algorithms.analysis.has_exact_type(a, ufl_type)
    Return if an object of class ufl_type can be found in a. The argument a can be a Form, Integral or Expr.

ufl.algorithms.analysis.has_type(a, ufl_type)
    Return if an object of class ufl_type can be found in a. The argument a can be a Form, Integral or Expr.

ufl.algorithms.analysis.sort_elements(elements)
    Sort elements so that any sub elements appear before the corresponding mixed elements. This is useful when
    sub elements need to be defined before the corresponding mixed elements.
    The ordering is based on sorting a directed acyclic graph.

ufl.algorithms.analysis.unique_tuple(objects)
    Return tuple of unique objects, preserving initial ordering.

ufl.algorithms.apply_algebra_lowering module

Algorithm for expanding compound expressions into equivalent representations using basic operators.

class ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra
    Bases: ufl.corealg.multifunction.MultiFunction

    Expands high level compound operators (e.g. inner) to equivalent representations using basic operators (e.g.
    index notation).

    alternative_dot(o, a, b)
    alternative_inner(o, a, b)
    cofactor(o, A)
    cross(o, a, b)
    curl(o, a)
    determinant(o, A)
    deviatoric(o, A)
    div(o, a)
    dot(o, a, b)
    expr(o, *ops)
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        expr = MultiFunction.reuse_if_untouched
        as a default rule.
    inner(o, a, b)
    inverse(o, A)
    nabla_div(o, a)
    nabla_grad(o, a)
**outer** \((o, a, b)\)
**skew** \((o, A)\)
**sym** \((o, A)\)
**trace** \((o, A)\)
**transposed** \((o, A)\)

`ufl.algorithms.apply_algebra_lowering.apply_algebra_lowering(expr)`
Expands high level compound operators (e.g. inner) to equivalent representations using basic operators (e.g. index notation).

**ufl.algorithms.apply_derivatives module**

This module contains the apply_derivatives algorithm which computes the derivatives of a form of expression.

```python
class ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher
    Bases: ufl.corealg.multifunction.MultiFunction

    coefficient_derivative \((o, f, dummy_w, dummy_v, dummy_cd)\)
    derivative \((o)\)
    expr \((o, *ops)\)
    Reuse object if operands are the same objects.
    Use in your own subclass by setting e.g.
    ```python
    expr = MultiFunction.reuse_if_untouched
    ```
    as a default rule.
    grad \((o, f)\)
    indexed \((o, Ap, ii)\)
    reference_grad \((o, f)\)
    terminal \((o)\)
    variable_derivative \((o, f, dummy_v)\)
```

```python
class ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset (coefficients, arguments, coefficient_derivatives)
    Bases: ufl.algorithms.apply_derivatives.GenericDerivativeRuleset

    Apply AFD (Automatic Functional Differentiation) to expression.
    Implements rules for the Gateaux derivative \(D_w[v](...)\) defined as
    ```math
    D_w[v](e) = \frac{d}{d\tau} e(w+\tau v)|_{\tau=0}
    ```
    argument \((o)\)
    Return a zero with the right shape for terminals independent of differentiation variable.
    cell_avg \((o, fp)\)
    coefficient \((o)\)
    facet_avg \((o, fp)\)
    geometric_quantity \((o)\)
    Return a zero with the right shape for terminals independent of differentiation variable.
```
grad(g)
reference_grad(o)
reference_value(o)

class ufl.algorithms.apply_derivatives.GenericDerivativeRuleset(var_shape)
    Bases: ufl.corealg.multifunction.MultiFunction

    abs(o, df)
    acos(o, fp)
    asin(o, fp)
    atan(o, fp)
    atan_2(o, fp, gp)
    bessel_i(o, nup, fp)
    bessel_j(o, nup, fp)
    bessel_k(o, nup, fp)
    bessel_y(o, nup, fp)
    binary_condition(o, dl, dr)
    cell_avg(o)
    component_tensor(o, Ap, ii)
    conditional(o, unused_dc, dt, df)
    constant_value(o)
    
    Return a zero with the right shape for terminals independent of differentiation variable.
    
    cos(o, fp)
    cosh(o, fp)
    derivative(o)
    division(o, fp, gp)
    erf(o, fp)
    exp(o, fp)
    expr(o)
    facet_avg(o)
    fixme(o)
    form_argument(o)
    geometric_quantity(o)
    grad(o)
    
    independent_operator(o)
    
    Return a zero with the right shape and indices for operators independent of differentiation variable.
    
    independent_terminal(o)
    
    Return a zero with the right shape for terminals independent of differentiation variable.
    
    index_sum(o, Ap, i)
indexed\( (o, Ap, ii) \)

label \( o \)

Labels and indices are not differentiable. It’s convenient to return the non-differentiated object.

list_tensor \( (o, *dops) \)

ln \( (o, fp) \)

math_function \( (o, df) \)

max_value \( (o, df, dg) \)

min_value \( (o, df, dg) \)

multi_index \( o \)

Labels and indices are not differentiable. It’s convenient to return the non-differentiated object.

non_differentiable_terminal \( o \)

Labels and indices are not differentiable. It’s convenient to return the non-differentiated object.

not_condition \( (o, c) \)

override \( o \)

power \( (o, fp, gp) \)

product \( (o, da, db) \)

restricted \( (o, fp) \)

sin \( (o, fp) \)

sinh \( (o, fp) \)

sqrt \( (o, fp) \)

sum \( (o, da, db) \)

tan \( (o, fp) \)

tanh \( (o, fp) \)

unexpected \( o \)

variable \( (o, df, unused_l) \)

class ufl.algorithms.apply_derivatives.GradRuleset \( (geometric\_dimension) \)

Bases: ufl.algorithms.apply_derivatives.GenericDerivativeRuleset

argument \( o \)

cell_avg \( o \)

Return a zero with the right shape and indices for operators independent of differentiation variable.

cell_coordinate \( o \)

dX/dx = inv(dx/dX) = inv(J) = K

coefficient \( o \)

cell_avg \( o \)

Return a zero with the right shape and indices for operators independent of differentiation variable.

geometric_quantity \( o \)

Default for geometric quantities is dg/dx = 0 if piecewise constant, otherwise keep Grad(g). Override for specific types if other behaviour is needed.
\begin{verbatim}
grad(o)
    Represent grad(grad(f)) as Grad(Grad(f)).

reference_grad(o)
reference_value(o)
spatial_coordinate(o)
    dx/dx = I

class ufl.algorithms.apply_derivatives.ReferenceGradRuleset (topological_dimension)
    Bases: ufl.algorithms.apply_derivatives.GenericDerivativeRuleset

    argument(o)
    cell_avg(o)
        Return a zero with the right shape and indices for operators independent of differentiation variable.
    cell_coordinate(o)
        dX/dX = I
    coefficient(o)
    facet_avg(o)
        Return a zero with the right shape and indices for operators independent of differentiation variable.
    geometric_quantity(o)
        dg/dX = 0 if piecewise constant, otherwise ReferenceGrad(g)
    grad(o)
    reference_grad(o)
        Represent ref_grad(ref_grad(f)) as RefGrad(RefGrad(f)).
    reference_value(o)
spatial_coordinate(o)
    dx/dX = J

class ufl.algorithms.apply_derivatives.VariableRuleset (var)
    Bases: ufl.algorithms.apply_derivatives.GenericDerivativeRuleset

    argument(o)
    cell_avg(o)
    cell_coordinate(o)
        Return a zero with the right shape for terminals independent of differentiation variable.
    coefficient(o)
        df/dv = Id if v is f else 0.
        Note that if v = variable(f), df/dv is still 0, but if v == f, i.e. isinstance(v, Coefficient) == True, then df/dv == df/df = Id.
    facet_avg(o)
    geometric_quantity(o)
        Return a zero with the right shape for terminals independent of differentiation variable.
    grad(o)
        Variable derivative of a gradient of a terminal must be 0.
    reference_grad(o)
        Variable derivative of a gradient of a terminal must be 0.
\end{verbatim}
reference_value(o)
variable(o, df, l)

ufl.algorithms.apply_derivatives.apply_derivatives(expression)

ufl.algorithms.apply_function_pullbacks module

Algorithm for replacing gradients in an expression with reference gradients and coordinate mappings.

class ufl.algorithms.apply_function_pullbacks.FunctionPullbackApplier
    Bases: ufl.corealg.multifunction.MultiFunction
    
expr(o, *ops)
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        
        expr = MultiFunction.reuse_if_untouched
        
as a default rule.
    
form_argument(o)
terminal(t)

ufl.algorithms.apply_function_pullbacks.apply_function_pullbacks(expr)
    Change representation of coefficients and arguments in expression by applying Piola mappings where applicable and representing all form arguments in reference value.

    @param expr: An Expr.

ufl.algorithms.apply_function_pullbacks.apply_single_function_pullbacks(g)
ufl.algorithms.apply_function_pullbacks.create_nested_lists(shape)
ufl.algorithms.apply_function_pullbacks.reshape_to_nested_list(components, shape)
ufl.algorithms.apply_function_pullbacks.sub_elements_with_mappings(element)

Return an ordered list of the largest subelements that have a defined mapping.

ufl.algorithms.apply_geometry_lowering module

Algorithm for lowering abstractions of geometric types.
This means replacing high-level types with expressions of mostly the Jacobian and reference cell data.

class ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier(preserve_types=())
    Bases: ufl.corealg.multifunction.MultiFunction
    
cell_coordinate(o)
cell_normal(o)
cell_volume(o)
circumradius(o)
expr(o, *ops)
    Reuse object if operands are the same objects.
    Use in your own subclass by setting e.g.
expr = MultiFunction.reuse_if_untouched

as a default rule.

facet_area(o)
facet_cell_coordinate(o)
facet_jacobian(o)
facet_jacobian_determinant(o)
facet_jacobian_inverse(o)
facet_normal(o)
jacobian(o)
jacobian_determinant(o)
jacobian_inverse(o)
max_cell_edge_length(o)
max_facet_edge_length(o)
min_cell_edge_length(o)
min_facet_edge_length(o)
spatial_coordinate(o)
terminal(t)

ufl.algorithms.apply_geometry_lowering.apply_geometry_lowering(form, preserve_types=())
Change GeometricQuantity objects in expression to the lowest level GeometricQuantity objects.
Assumes the expression is preprocessed or at least that derivatives have been expanded.

@param form: An Expr or Form.

ufl.algorithms.apply_integral_scaling module

Algorithm for replacing gradients in an expression with reference gradients and coordinate mappings.

ufl.algorithms.apply_integral_scaling.apply_integral_scaling(form)
Multiply integrands by a factor to scale the integral to reference frame.

ufl.algorithms.apply_integral_scaling.compute_integrand_scaling_factor(integral)
Change integrand geometry to the right representations.

ufl.algorithms.apply_restrictions module

This module contains the apply_restrictions algorithm which propagates restrictions in a form towards the terminals.

class ufl.algorithms.apply_restrictions.RestrictionPropagator(side=None)
    Bases: ufl.corealg.multifunction.MultiFunction

    argument(o)
    Restrict a discontinuous quantity to current side, require a side to be set.

    cell_avg(o)
    Restrict a discontinuous quantity to current side, require a side to be set.

2.2. ufl package
cell_coordinate

Restrict a discontinuous quantity to current side, require a side to be set.

cell_edge_vectors

Restrict a discontinuous quantity to current side, require a side to be set.

cell_facet_jacobian

Restrict a discontinuous quantity to current side, require a side to be set.

cell_facet_jacobian_determinant

Restrict a discontinuous quantity to current side, require a side to be set.

cell_facet_jacobian_inverse

Restrict a discontinuous quantity to current side, require a side to be set.

cell_facet_origin

Restrict a discontinuous quantity to current side, require a side to be set.

cell_normal

Restrict a discontinuous quantity to current side, require a side to be set.

cell_orientation

Restrict a discontinuous quantity to current side, require a side to be set.

cell_origin

Restrict a discontinuous quantity to current side, require a side to be set.

cell_volume

Restrict a discontinuous quantity to current side, require a side to be set.

circumradius

Restrict a discontinuous quantity to current side, require a side to be set.

coefficient

Allow coefficients to be unrestricted (apply default if so) if the values are fully continuous across the facet.

facet_area

Restrict a continuous quantity to default side if no current restriction is set.

facet_avg

Ignore current restriction, quantity is independent of side also from a computational point of view.

facet_coordinate

Ignore current restriction, quantity is independent of side also from a computational point of view.

facet_jacobian

Restrict a continuous quantity to default side if no current restriction is set.

facet_jacobian_determinant

Restrict a continuous quantity to default side if no current restriction is set.

facet_jacobian_inverse

Restrict a continuous quantity to default side if no current restriction is set.

facet_normal

Restrict a discontinuous quantity to current side, require a side to be set.

facet_origin

Restrict a continuous quantity to default side if no current restriction is set.

terrestrial_cell_quantity

terrestrial_facet_quantity
\texttt{grad} \( (o) \) 
Restrict a discontinuous quantity to current side, require a side to be set.

\texttt{jacobian} \( (o) \) 
Restrict a discontinuous quantity to current side, require a side to be set.

\texttt{jacobian\_determinant} \( (o) \) 
Restrict a discontinuous quantity to current side, require a side to be set.

\texttt{jacobian\_inverse} \( (o) \) 
Restrict a discontinuous quantity to current side, require a side to be set.

\texttt{max\_facet\_edge\_length} \( (o) \) 
Restrict a continuous quantity to default side if no current restriction is set.

\texttt{min\_facet\_edge\_length} \( (o) \) 
Restrict a continuous quantity to default side if no current restriction is set.

\texttt{operator} \( (o, \ast \text{ops}) \) 
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

\begin{verbatim}
expr = MultiFunction.reuse_if_untouched
\end{verbatim}
as a default rule.

\texttt{quadrature\_weight} \( (o) \) 
Ignore current restriction, quantity is independent of side also from a computational point of view.

\texttt{reference\_normal} \( (o) \)

\texttt{reference\_value} \( (o) \) 
Reference value of something follows same restriction rule as the underlying object.

\texttt{restricted} \( (o) \) 
When hitting a restricted quantity, visit child with a separate restriction algorithm.

\texttt{spatial\_coordinate} \( (o) \) 
Restrict a continuous quantity to default side if no current restriction is set.

\texttt{terminal} \( (o) \) 
Ignore current restriction, quantity is independent of side also from a computational point of view.

\texttt{variable} \( (o, \text{op}) \) 
Strip variable.

\texttt{ufl.algorithms.apply\_restrictions.apply\_restrictions} \( (\text{expression}) \) 
Propagate restriction nodes to wrap differential terminals directly.

\texttt{ufl.algorithms.argument\_dependencies module}

Algorithms for analysing argument dependencies in expressions.

\texttt{class ufl.algorithms.argument\_dependencies.ArgumentDependencyExtractor} 
Bases: \texttt{ufl.algorithms.transformer.Transformer}

\texttt{argument} \( (o) \)

\texttt{cell\_avg} \( (o, a) \) 
Nonterminals that are linear with a single argument.

\texttt{component\_tensor} \( (o, f, i) \)
conditional \((o, \text{cond}, t, f)\)
   Considering EQ, NE, LE, GE, LT, GT nonlinear in this context.

cross \((o, *\text{opdeps})\)

curl \((o, a)\)
   Nonterminals that are linear with a single argument.

div \((o, a)\)
   Nonterminals that are linear with a single argument.

division \((o, a, b)\)
   Arguments cannot be in the denominator.

dot \((o, *\text{opdeps})\)

facet_avg \((o, a)\)
   Nonterminals that are linear with a single argument.

grad \((o, a)\)
   Nonterminals that are linear with a single argument.

index_sum \((o, f, i)\)
   Index sums inherit the dependencies of their summand.

indexed \((o, f, i)\)

inner \((o, *\text{opdeps})\)

linear \((o, a)\)
   Nonterminals that are linear with a single argument.

list_tensor \((o, *\text{opdeps})\)
   Require same dependencies for all listtensor entries.

max_value \((o, l, r)\)
   Considering min, max nonlinear in this context.

min_value \((o, l, r)\)
   Considering min, max nonlinear in this context.

nabla_div \((o, a)\)
   Nonterminals that are linear with a single argument.

nabla_grad \((o, a)\)
   Nonterminals that are linear with a single argument.

negative_restricted \((o, a)\)
   Nonterminals that are linear with a single argument.

operator \((o, *\text{opdeps})\)
   Default for Operators: nonlinear in all operands.

outer \((o, *\text{opdeps})\)

positive_restricted \((o, a)\)
   Nonterminals that are linear with a single argument.

product \((o, *\text{opdeps})\)

reference_curl \((o, a)\)
   Nonterminals that are linear with a single argument.

reference_div \((o, a)\)
   Nonterminals that are linear with a single argument.
**reference_grad**(*o, a*)
Nonterminals that are linear with a single argument.

**skew**(*o, a*)
Nonterminals that are linear with a single argument.

**spatial_derivative**(*o, a, b*)

**sum**(*o, *opdeps*)
Sums can contain both linear and bilinear terms (we could change this to require that all operands have the same dependencies).

**sym**(*o, a*)
Nonterminals that are linear with a single argument.

**terminal**(*o*)
Default for terminals: no dependency on Arguments.

**trace**(*o, a*)
Nonterminals that are linear with a single argument.

**transposed**(*o, a*)
Nonterminals that are linear with a single argument.

**variable**(*o*)

**variable_derivative**(*o, a, b*)

---

**exception** ufl.algorithms.argument_dependencies.NotMultiLinearException(*args, **kwargs*)

Bases: exceptions.Exception

ufl.algorithms.argument_dependencies.extract_argument_dependencies(*e*)
Extract a set of sets of Arguments.

---

**ufl.algorithms.change_to_reference module**

Algorithm for replacing gradients in an expression with reference gradients and coordinate mappings.

**class** ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad
Bases: ufl.corealg.multifunction.MultiFunction

**cell_avg**(*o, *dummy_ops*)

**coefficient_derivative**(*o, *dummy_ops*)

**expr**(*o, *ops*)

**facet_avg**(*o, *dummy_ops*)

**form_argument**(*t*)

**geometric_quantity**(*t*)

**grad**(*o, *dummy_ops*)
Store modifier state.

**reference_grad**(*o, *dummy_ops*)

**restricted**(*o, *dummy_ops*)
Store modifier state.

**terminal**(*o*)
class ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad
    Bases: ufl.corealg.multifunction.MultiFunction

    coefficient_derivative(o)
    expr(o, *ops)
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        expr = MultiFunction.reuse_if_untouched
        as a default rule.
    grad(o)
    reference_grad(o)
    terminal(o)

ufl.algorithms.change_to_reference.change_integrand_geometry_representation(integrand, scale, integral_type)

    Change integrand geometry to the right representations.

ufl.algorithms.change_to_reference.change_to_reference_grad(e)
    Change Grad objects in expression to products of JacobianInverse and ReferenceGrad.
    Assumes the expression is preprocessed or at least that derivatives have been expanded.
    @param e: An Expr or Form.

ufl.algorithms.check_arities module

class ufl.algorithms.check_arities.ArityChecker(arguments)
    Bases: ufl.corealg.multifunction.MultiFunction

    argument(o)
    cell_avg(o, a)
    component_tensor(o, a, i)
    division(o, a, b)
    dot(o, a, b)
    expr(o)
    facet_avg(o, a)
    grad(o, a)
    index_sum(o, a, i)
    indexed(o, a, i)
    inner(o, a, b)
    linear_indexed_type(o, a, i)
    linear_operator(o, a)
    list_tensor(o, *ops)
negative_restricted $(o, a)$
nonlinear_operator $(o)$
outer $(o, a, b)$
positive_restricted $(o, a)$
product $(o, a, b)$
reference_grad $(o, a)$
reference_value $(o, a)$
sum $(o, a, b)$
terminal $(o)$
variable $(o, f, l)$

exception ufl.algorithms.check_arities.ArrityMismatch
    Bases: ufl.log.UFLException

ufl.algorithms.check_arities.check_form_arity $(form, arguments)$
ufl.algorithms.check_arities.check_integrand_arity $(expr, arguments)$

ufl.algorithms.check_restrictions module

Algorithms related to restrictions.
class ufl.algorithms.check_restrictions.RestrictionChecker $(require_restriction)$
    Bases: ufl.algorithms.transformer.Transformer
        expr $(o)$
        facet_normal $(o)$
        form_argument $(o)$
        restricted $(o)$
ufl.algorithms.check_restrictions.check_restrictions $(expression, require_restriction)$

ufl.algorithms.checks module

Functions to check the validity of forms.
ufl.algorithms.checks.validate_form $(form)$
    Performs all implemented validations on a form. Raises exception if something fails.

ufl.algorithms.compute_form_data module

This module provides the compute_form_data function which form compilers will typically call prior to code generation to preprocess/simplify a raw input form given by a user.
ufl.algorithms.compute_form_data.compute_form_data(form, 
   do_apply_function_pullbacks=False, 
   do_apply_integral_scaling=False, 
   do_apply_geometry_lowering=False, 
   preserve_geometry_types=(), 
   do_apply_restrictions=True)

ufl.algorithms.domain_analysis module

Algorithms for building canonical data structure for integrals over subdomains.

class ufl.algorithms.domain_analysis.ExprTupleKey(x)
   Bases: object
   x

class ufl.algorithms.domain_analysis.IntegralData(domain, integral_type, subdomain_id, integrals, metadata)
   Bases: object
   Utility class with the members (domain, integral_type, subdomain_id, integrals, metadata)
   where metadata is an empty dictionary that may be used for associating metadata with each object.
   
   domain
   enabled_coefficients
   integral_coefficients
   integral_type
   integrals
   metadata
   subdomain_id

ufl.algorithms.domain_analysis.accumulate_integrands_with_same_metadata(integrals)
   Taking input on the form: integrals = [integral0, integral1, ...]
   Return result on the form:
   integrands_by_id = [(integrand0, metadata0), (integrand1, metadata1), ...]
   where integrand0 < integrand1 by the canonical ufl expression ordering criteria.

ufl.algorithms.domain_analysis.build_integral_data(integrals, domains)

ufl.algorithms.domain_analysis.dicts_lt(a, b)

ufl.algorithms.domain_analysis.group_integrals_by_domain_and_type(integrals, domains)
   Input: integrals: list of Integral objects domains: list of AbstractDomain objects from the parent Form
   Output: integrals_by_domain_and_type: dict: (domain, integral_type) -> list(Integral)

ufl.algorithms.domain_analysis.integral_subdomain_ids(integral)
   Get a tuple of integer subdomains or a valid string subdomain from integral.

ufl.algorithms.domain_analysis.rearrange_integrals_by_single_subdomains(integrals)
   Rearrange integrals over multiple subdomains to single subdomain integrals.
   Input: integrals: list(Integral)
Output: integrals: dict: subdomain_id -> list(Integral) (reconstructed with single subdomain_id)

```
ufl.algorithms.domain_analysis.reconstruct_form_from_integral_data(integral_data)
```

**ufl.algorithms.elementtransformations module**

```
ufl.algorithms.elementtransformations.change_regularity(element, family)
```

For a given finite element, return the corresponding space specified by ‘family’.

```
ufl.algorithms.elementtransformations.increase_order(element)
```

Return element of same family, but a polynomial degree higher.

```
ufl.algorithms.elementtransformations.tear(V)
```

For a finite element, return the corresponding discontinuous element.

**ufl.algorithms.estimate_degrees module**

Algorithms for estimating polynomial degrees of expressions.

```
class ufl.algorithms.estimate_degrees.SumDegreeEstimator(default_degree, element_replace_map)
```

```
Bases: ufl.algorithms.transformer.Transformer
```

This algorithm is exact for a few operators and heuristic for many.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(v, a)</code></td>
<td>This is a heuristic, correct if there is no argument</td>
</tr>
<tr>
<td><code>argument(v)</code></td>
<td>A form argument provides a degree depending on the element, or the default degree if the element has no degree.</td>
</tr>
<tr>
<td><code>atan_2(v, a, b)</code></td>
<td>Using the heuristic degree(atan2(const,const)) == 0 degree(atan2(a,b)) == max(degree(a),degree(b))+2 which can be wildly inaccurate but at least gives a somewhat high integration degree.</td>
</tr>
<tr>
<td><code>bessel_function(v, nu, x)</code></td>
<td>Using the heuristic degree(bessel_* (const)) == 0 degree(bessel_* (x)) == degree(x)+2 which can be wildly inaccurate but at least gives a somewhat high integration degree.</td>
</tr>
<tr>
<td><code>cell_avg(v, a)</code></td>
<td>Cell average of a function is always cellwise constant.</td>
</tr>
<tr>
<td><code>cell_coordinate(v)</code></td>
<td>A coordinate provides one additional degree.</td>
</tr>
<tr>
<td><code>coefficient(v)</code></td>
<td>A form argument provides a degree depending on the element, or the default degree if the element has no degree.</td>
</tr>
<tr>
<td><code>cofactor(v, *args)</code></td>
<td></td>
</tr>
<tr>
<td><code>compound_tensor(v, A, ii)</code></td>
<td></td>
</tr>
<tr>
<td><code>compound_derivative(v, *args)</code></td>
<td></td>
</tr>
<tr>
<td><code>compound_tensor_operator(v, *args)</code></td>
<td></td>
</tr>
<tr>
<td><code>condition(v, *args)</code></td>
<td></td>
</tr>
</tbody>
</table>
conditional \((v, c, t, f)\)
Degree of condition does not influence degree of values which conditional takes. So heuristically taking max of true degree and false degree. This will be exact in cells where condition takes single value. For improving accuracy of quadrature near condition transition surface quadrature order must be adjusted manually.

constant_value \((v)\)
Constant values are constant.

cross \((v, \ast \text{ops})\)


curl \((v, f)\)
Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

derivative \((v, \ast \text{args})\)

determinant \((v, \ast \text{args})\)

deviatoric \((v, \ast \text{args})\)

div \((v, f)\)
Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

division \((v, \ast \text{ops})\)
Using the sum here is a heuristic. Consider e.g. \((x+1)/(x-1)\).

dot \((v, \ast \text{ops})\)

expr \((v, \ast \text{ops})\)
For most operators we take the max degree of its operands.

facet_avg \((v, a)\)
Facet average of a function is always cellwise constant.

geometric_quantity \((v)\)
Some geometric quantities are cellwise constant. Others are nonpolynomial and thus hard to estimate.

grad \((v, f)\)
Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

index_sum \((v, A, ii)\)

indexed \((v, A, ii)\)

inner \((v, \ast \text{ops})\)

inverse \((v, \ast \text{args})\)

label \((v)\)

list_tensor \((v, \ast \text{ops})\)

math_function \((v, a)\)
Using the heuristic degree(sin(const)) == 0 degree(sin(a)) == degree(a)+2 which can be wildly inaccurate but at least gives a somewhat high integration degree.

max_value \((v, l, r)\)
Same as conditional.

min_value \((v, l, r)\)
Same as conditional.
**multi_index**(v)

**nabla_div**(v,f)
   Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

**nabla_grad**(v,f)
   Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

**negative_restricted**(v, a)

**outer**(v, *ops)

**positive_restricted**(v, a)

**power**(v, a, b)
   If b is an integer: degree(a**b) == degree(a)*b otherwise use the heuristic degree(a**b) == degree(a)*2

**product**(v, *ops)

**reference_curl**(v,f)
   Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

**reference_div**(v,f)
   Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

**reference_grad**(v,f)
   Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when OuterProduct elements are involved.

**reference_value**(rv,f)

**skew**(v, *args)

**spatial_coordinate**(v)
   A coordinate provides additional degrees depending on coordinate field of domain.

**sum**(v, *ops)

**sym**(v, *args)

**trace**(v, *args)

**transposed**(v, A)

**variable**(v, e, l)

**variable_derivative**(v, *args)

**ufl.algorithms.estimate_degrees.estimate_total_polynomial_degree**(e, default_degree=1, element_replace_map={})

Estimate total polynomial degree of integrand.

NB! Although some compound types are supported here, some derivatives and compounds must be preprocessed prior to degree estimation. In generic code, this algorithm should only be applied after preprocessing.

For coefficients defined on an element with unspecified degree (None), the degree is set to the given default degree.
ufl.algorithms.expand_compounds module

Algorithm for expanding compound expressions into equivalent representations using basic operators.

```python
ufl.algorithms.expand_compounds.expand_compounds(e)
```

ufl.algorithms.expand_indices module

This module defines expression transformation utilities, for expanding free indices in expressions to explicit fixed indices only.

```python
class ufl.algorithms.expand_indices.IndexExpander
    Bases: ufl.algorithms.transformer.ReuseTransformer

    component()
        Return current component tuple.

    component_tensor(x)

    division(x)

    form_argument(x)

    grad(x)

    index_sum(x)

    indexed(x)

    list_tensor(x)

    multi_index(x)

    scalar_value(x)

    terminal(x)

    zero(x)
```

```python
ufl.algorithms.expand_indices.expand_indices(e)
```

```python
ufl.algorithms.expand_indices.purge_list_tensors(expr)
```

Get rid of all ListTensor instances by expanding expressions to use their components directly. Will usually increase the size of the expression.

ufl.algorithms.formdata module

FormData class easy for collecting of various data about a form.

```python
class ufl.algorithms.formdata.ExprData
    Bases: object

    Class collecting various information extracted from a Expr by calling preprocess.
```

```python
class ufl.algorithms.formdata.FormData
    Bases: object

    Class collecting various information extracted from a Form by calling preprocess.
```
ufl.algorithms.formfiles module

A collection of utility algorithms for handling UFL files.

```python
class ufl.algorithms.formfiles.FileData
    Bases: object

ufl.algorithms.formfiles.execute_ufl_code(uflcode, filename)

ufl.algorithms.formfiles.interpret_ufl_namespace(namespace)
    Takes a namespace dict from an executed ufl file and converts it to a FileData object.

ufl.algorithms.formfiles.load_forms(filename)
    Return a list of all forms in a file.

ufl.algorithms.formfiles.load_ufl_file(filename)
    Load a .ufl file with elements, coefficients and forms.

ufl.algorithms.formfiles.read_ufl_file(filename)
    Read a ufl file, handling file extension, file existance, and #include replacement.

ufl.algorithms.formfiles.replace_include_statements(code)
    Replace ‘#include foo.ufl’ statements with contents of foo.ufl.
```

ufl.algorithms.formtransformations module

This module defines utilities for transforming complete Forms into new related Forms.

```python
class ufl.algorithms.formtransformations.PartExtracter (arguments)
    Bases: ufl.algorithms.transformer.Transformer

PartExtracter extracts those parts of a form that contain the given argument(s).

argument (x)
    Return itself unless itself provides too much.

cell_avg (x, arg)
    A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

component_tensor (x)
    Return parts of expression belonging to this indexed expression.

division (x)
    Return parts_of_numerator/denominator.

dot (x, *ops)
    Note: Product is a visit-children-first handler. ops are the visited factors.

expr (x)
    The default is a nonlinear operator not accepting any Arguments among its children.

facet_avg (x, arg)
    A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

grad (x, arg)
    A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

index_sum (x)
    Return parts of expression belonging to this indexed expression.

indexed (x)
    Return parts of expression belonging to this indexed expression.
```
inner \((x, *ops)\)
Note: Product is a visit-children-first handler. ops are the visited factors.

linear_indexed_type \((x)\)
Return parts of expression belonging to this indexed expression.

linear_operator \((x, arg)\)
A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

list_tensor \((x, *ops)\)

negative_restricted \((x, arg)\)
A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

outer \((x, *ops)\)
Note: Product is a visit-children-first handler. ops are the visited factors.

positive_restricted \((x, arg)\)
A linear operator with a single operand accepting arity > 0, providing whatever Argument its operand does.

product \((x, *ops)\)
Note: Product is a visit-children-first handler. ops are the visited factors.

sum \((x)\)
Return the terms that might eventually yield the correct parts(!)

The logic required for sums is a bit elaborate:

A sum may contain terms providing different arguments. We should return (a sum of) a suitable subset of these terms. Those should all provide the same arguments.

For each term in a sum, there are 2 simple possibilities:
1a) The relevant part of the term is zero -> skip. 1b) The term provides more arguments than we want -> skip
2) If all terms fall into the above category, we can just return zero.

Any remaining terms may provide exactly the arguments we want, or fewer. This is where things start getting interesting.

3) Bottom-line: if there are terms with providing different arguments – provide terms that contain the most arguments. If there are terms providing different sets of same size -> throw error (e.g. Argument(-1) + Argument(-2))

terminal \((x)\)
The default is a nonlinear operator not accepting any Arguments among its children.

variable \((x)\)
Return relevant parts of this variable.

`ufl.algorithms.formtransformations.compute_energy_norm(form, coefficient)`
Compute the a-norm of a Coefficient given a form a.

This works simply by replacing the two Arguments with a Coefficient on the same function space (element). The Form returned will thus be a functional with no Arguments, and one additional Coefficient at the end if no coefficient has been provided.

`ufl.algorithms.formtransformations.compute_form_action(form, coefficient)`
Compute the action of a form on a Coefficient.

This works simply by replacing the last Argument with a Coefficient on the same function space (element). The form returned will thus have one Argument less and one additional Coefficient at the end if no Coefficient has been provided.
ufl.algorithms.formtransformations.compute_form_adjoint(form, reordered_arguments=None)

Compute the adjoint of a bilinear form.
This works simply by swapping the number and part of the two arguments, but keeping their elements and places in the integrand expressions.

ufl.algorithms.formtransformations.compute_form_arities(form)

Return set of arities of terms present in form.

ufl.algorithms.formtransformations.compute_form_functional(form)

Compute the functional part of a form, that is the terms independent of Arguments.
(Used for testing, not sure if it’s useful for anything?)

ufl.algorithms.formtransformations.compute_form_lhs(form)

Compute the left hand side of a form.
Example:
\[ a = u*v*dx + f*v*dx \]
a = \text{lhs}(a) \rightarrow u*v*dx

ufl.algorithms.formtransformations.compute_form_rhs(form)

Compute the right hand side of a form.
Example:
\[ a = u*v*dx + f*v*dx \]
L = \text{rhs}(a) \rightarrow -f*v*dx

ufl.algorithms.formtransformations.compute_form_with_arity(form, arity, arguments=None)

Compute parts of form of given arity.

ufl.algorithms.formtransformations.zero_expr(e)

ufl.algorithms.forward_ad module

Forward mode AD implementation.

class ufl.algorithms.forward_ad.CoefficientAD(coefficients, arguments, coefficient_derivatives, cache=None)

Apply AFD (Automatic Functional Differentiation) to expression.

coefficient(o)

grad(g)

variable(o)

class ufl.algorithms.forward_ad.ForwardAD(var_shape, cache=None)

Bases: ufl.algorithms.formtransformations.Transformer

abs(o, a)

acos(o, a)

asin(o, a)

atan(o, a)

atan_2(o, a, b)

bessel_i(o, nu, x)
The UFL API includes a wide range of mathematical functions and operations:

- `bessel_j(o, nu, x)`
- `bessel_k(o, nu, x)`
- `bessel_y(o, nu, x)`
- `binary_condition(o, l, r)`
- `cell_avg(o, a)`
- `component_tensor(o)`
- `conditional(o, c, t, f)`
- `cos(o, a)`
- `cosh(o, a)`
- `derivative(o)`
- `division(o, a, b)`
- `erf(o, a)`
- `exp(o, a)`
- `expr(o)`
- `facet_avg(o, a)`
- `grad(o)`
- `index_sum(o)`
- `indexed(o)`
- `list_tensor(o, *ops)`
- `ln(o, a)`
- `math_function(o, a)`
- `max_value(o, x, y)`
- `min_value(o, x, y)`
- `multi_index(o)`
- `not_condition(o, c)`
- `power(o, a, b)`
- `product(o, *ops)`
- `restricted(o, a)`
- `sin(o, a)`
- `sinh(o, a)`
- `sqrt(o, a)`
- `sum(o, *ops)`
- `tan(o, a)`
- `tanh(o, a)`
- `terminal(o)`

Terminal objects are assumed independent of the differentiation variable by default, and simply 'lifted' to the pair (o, 0). Depending on the context, override this with custom rules for non-zero derivatives.
variable (o)
Variable objects are just ‘labels’, so by default the derivative of a variable is the derivative of its referenced expression.

class ufl.algorithms.forward_ad.GradAD (geometric_dimension, cache=None)
Bases: ufl.algorithms.forward_ad.ForwardAD

argument (o)
Represent \( \text{grad}(f) \) as \( \text{Grad}(f) \).

cell_coordinate (o)
Gradient of \( X \) w.r.t. \( x \) is \( K \). But I’m not sure if we want to allow this.

coefficient (o)
Represent \( \text{grad}(f) \) as \( \text{Grad}(f) \).

facet_coordinate (o)

facet_jacobian (o)

facet_jacobian_determinant (o)

facet_jacobian_inverse (o)

geometric_quantity (o)
Represent \( \text{grad}(g) \) as \( \text{Grad}(g) \).

grad (o)
Represent \( \text{grad}(\text{grad}(f)) \) as \( \text{Grad}(\text{Grad}(f)) \).

jacobian (o)

jacobian_determinant (o)

jacobian_inverse (o)

spatial_coordinate (o)
Gradient of \( x \) w.r.t. \( x \) is \( \text{Id} \).

class ufl.algorithms.forward_ad.UnimplementedADRules
Bases: object

cofactor (o, a)

cross (o, a, b)

determinant (o, a)

FIXME: Some possible rules:

\[
\frac{d \det A}{dv} = \det A \cdot \text{tr}(\text{inv}(A) \cdot \frac{dA}{dv})
\]

or

\[
\frac{d \det A}{d \text{row0}} = \text{cross}(\text{row1}, \text{row2}) \quad \frac{d \det A}{d \text{row1}} = \text{cross}(\text{row2}, \text{row0}) \quad \frac{d \det A}{d \text{row2}} = \text{cross}(\text{row0}, \text{row1})
\]

i.e.

\[
\frac{d \det A}{d A} = \left[ \text{cross}(\text{row1}, \text{row2}), \text{cross}(\text{row2}, \text{row0}), \text{cross}(\text{row0}, \text{row1}) \right] \text{ or transposed or something}
\]

inverse (o, a)

Derivation: \( 0 = \frac{d}{dx} [\text{Ainv}^*A] = \text{Ainv}' \cdot A + \text{Ainv} \cdot A' \text{Ainv}' \cdot A = - \text{Ainv} \cdot A' \text{Ainv}' = - \text{Ainv} \cdot A' \cdot \text{Ainv} \)
\[
\]

**class** ufl.algorithms.forward_ad.UnusedADRules

**Bases:** object

**commute** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**curl** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**deviatoric** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**div** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**dot** \((o, a, b)\)

**grad** \((o, a)\)

**inner** \((o, a, b)\)

**outer** \((o, a, b)\)

**trace** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**transposed** \((o, a)\)

This should work for all single argument operators that commute with \(d/dw\) with \(w\) scalar.

**class** ufl.algorithms.forward_ad.VariableAD \((var, cache=None)\)

**Bases:** ufl.algorithms.forward_ad.ForwardAD

**grad** \((o)\)

**variable** \((o)\)

ufl.algorithms.forward_ad.apply_nested_forward_ad(expr)

ufl.algorithms.forward_ad.compute_coefficient_forward_ad(f, w, v, cd)

ufl.algorithms.forward_ad.compute_grad_forward_ad(f, geometric_dimension)

ufl.algorithms.forward_ad.compute_variable_forward_ad(f, v)

**ufl.algorithms.map_integrands module**

Basic algorithms for applying functions to subexpressions.

**ufl.algorithms.map_integrands.map_integrand_dags** \(function\), \(form\), \(only\_integral\_type=None\), \(compress=True\)

**ufl.algorithms.map_integrands.map_integrands** \(function\), \(form\), \(only\_integral\_type=None\)

Apply transform(expression) to each integrand expression in form, or to form if it is an Expr.
This module defines partial differentiation rules for all relevant operands for use with reverse mode AD.

```python
class ufl.algorithms.pdiffs.PartialDerivativeComputer
    Bases: ufl.corealg.multifunction.MultiFunction
```

NB! The main reason for keeping this out of the Expr hierarchy is to avoid user mistakes in the form of mixups with total derivatives, and to allow both reverse and forward mode AD.

```
abs(f)

fracdxf(x) =
fracdxb(x) = sign(x)
```

```
acos(f)

d/dx arccos(x) = -1 / \sqrt{1 - x^2}
```

```
asin(f)

d/dx asin x = 1 / sqrt(1 - x^2)
```

```
atan(f)

d/dx atan x = 1 / (1 + x^2)
```

```
atan_2(f)

f = atan2(x,y)
d/dx atan2(x,y) = y / (x^2 + y^2)
d/dy atan2(x,y) = -x / (x^2 + y^2)
```

```
bessel_function(nu, x)
```

```
cell_avg(f)
```

```
coefficient_derivative(f)
```

```
component_tensor(f)
```

```
condition(f)
```

```
conditional(f)
```

```
cos(f)

d/dx cos x = -sin(x)
```

```
cosh(f)

d/dx cosh x = sinh(x)
```

```
division(f)

f = x/y
d/dx x/y = 1/y
d/dy x/y = -x/y^2 = -f/y
```

```
erf(f)

d/dx erf x = 2/sqrt(pi)*exp(-x^2)
```

```
exp(f)

d/dx exp(x) = exp(x)
```

```
expr(o)
```
**facet_avg** *(f)*

**index_sum** *(f)*
\[ \frac{d}{dx} \sum_j x = \text{TODO} \]

**indexed** *(f)*
\[ \frac{d}{dx} x_i = (1)_i = 1 \]

**list_tensor** *(f)*
\[ \frac{d}{dx_i} [x_0, ..., x_{n-1}] = e_i \text{ (unit vector)} \]

**ln** *(f)*
\[ \frac{d}{dx} \ln x = \frac{1}{x} \]

**negative_restricted** *(f)*

**positive_restricted** *(f)*

**power** *(f)*
\[ f = x^{**y} \quad \frac{d}{dx} x^{**y} = \frac{y*x^{**(y-1)} = y*f/x} \]

**product** *(f)*

**sin** *(f)*
\[ \frac{d}{dx} \sin x = \cos(x) \]

**sinh** *(f)*
\[ \frac{d}{dx} \sinh x = \cosh(x) \]

**spatial_derivative** *(f)*

**sqrt** *(f)*
\[ \frac{d}{dx} \sqrt{x} = \frac{1}{2*\sqrt{x}} \]

**sum** *(f)*
\[ \frac{d}{dx_i} \sum_j x_j = 1 \]

**tan** *(f)*
\[ \frac{d}{dx} \tan x = (\sec(x))^2 = \frac{2}{\cos(2x) + 1} \]

**tanh** *(f)*
\[ \frac{d}{dx} \tanh x = (\sech(x))^2 = \left(\frac{2 \cosh(x)}{(\cosh(2x) + 1)}\right)^2 \]

**variable_derivative** *(f)*

ufl.algorithms.pdiffs.pdiffs*(exprs)*

**ufl.algorithms.predicates module**

Functions to check properties of forms and integrals.

ufl.algorithms.predicates.is_multilinear*(form)*

Check if form is multilinear in arguments.

**ufl.algorithms.renumbering module**

Algorithms for renumbering of counted objects, currently variables and indices.

class ufl.algorithms.renumbering.IndexRenumberingTransformer

**Bases:** ufl.algorithms.renumbering.VariableRenumberingTransformer

This is a poorly designed algorithm. It is used in some tests, please do not use for anything else.
index(o)
multi_index(o)
zero(o)

class ufl.algorithms.renumbering.VariableRenumberingTransformer
Bases: ufl.algorithms.transformer.ReuseTransformer

class ufl.algorithms.renumbering.renumber_indices(expr)

ufl.algorithms.renumber module

Algorithm for replacing terminals in an expression.

class ufl.algorithms.replace.Replacer(mapping)
Bases: ufl.corealg.multifunction.MultiFunction

coefficient_derivative(o)
expr(o, *ops)
    Reuse object if operands are the same objects.
    Use in your own subclass by setting e.g.
    expr = MultiFunction.reuse_if_untouched
as a default rule.

terminal(o)

ufl.algorithms.replace.replace(e, mapping)
Replace terminal objects in expression.

@param e: An Expr or Form.

@param mapping: A dict with from:to replacements to perform.

ufl.algorithms.signature module

Signature computation for forms.

ufl.algorithms.signature.compute_expression_hashdata(expression, terminal_hashdata)
ufl.algorithms.signature.compute_expression_signature(expr, renumbering)
ufl.algorithms.signature.compute_form_signature(form, renumbering)
ufl.algorithms.signature.compute_multiindex_hashdata(expr, index_numbering)
ufl.algorithms.signature.compute_terminal_hashdata(expressions, renumbering)

ufl.algorithms.transformer module

This module defines the Transformer base class and some basic specializations to further base other algorithms upon, as well as some utilities for easier application of such algorithms.

class ufl.algorithms.transformer.CopyTransformer(variable_cache=None)
Bases: ufl.algorithms.transformer.Transformer
expr \,(o, \ast\,operands)\]
    Always reconstruct expr.

terminal \,(o)\]
    Always reuse Expr (ignore children)

variable \,(o)\]

class ufl.algorithms.transformer.ReuseTransformer (variable_cache=None)
    Bases: ufl.algorithms.transformer.Transformer

    expr \,(o, \ast\,ops)\]
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        
        expr = MultiFunction.reuse_if_untouched
        
        as a default rule.

    terminal \,(o)\]
        Always reuse Expr (ignore children)

variable \,(o)\]

class ufl.algorithms.transformer.Transformer (variable_cache=None)
    Bases: object

    Base class for a visitor-like algorithm design pattern used to transform expression trees from one representation to another.

    always_reconstruct \,(o, \ast\,operands)\]
        Always reconstruct expr.

    expr \,(o)\]
        Trigger error.

    print_visit_stack ()

    reconstruct_variable \,(o)\]

    reuse \,(o)\]
        Always reuse Expr (ignore children)

    reuse_if_possible \,(o, \ast\,ops)\]
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        
        expr = MultiFunction.reuse_if_untouched
        
        as a default rule.

    reuse_if_untouched \,(o, \ast\,ops)\]
        Reuse object if operands are the same objects.
        Use in your own subclass by setting e.g.
        
        expr = MultiFunction.reuse_if_untouched
        
        as a default rule.

    reuse_variable \,(o)\]

    terminal \,(o)\]
        Always reuse Expr (ignore children)
undefined(o)
Trigger error.

visit(o)

class ufl.algorithms.transformer.VariableStripper
    Bases: ufl.algorithms.transformer.ReuseTransformer

variable(o)

ufl.algorithms.transformer.apply_transformer(e, transformer, integral_type=None)
    Apply transformer.visit(expression) to each integrand expression in form, or to form if it is an Expr.

ufl.algorithms.transformer.is_post_handler(function)
    Is this a handler that expects transformed children as input?

ufl.algorithms.transformer.strip_variables(e)
    Replace all Variable instances with the expression they represent.

ufl.algorithms.transformer.ufl2ufl(e)
    Convert an UFL expression to a new UFL expression, with no changes. This is used for testing that objects in the expression behave as expected.

ufl.algorithms.transformer.ufl2uflcopy(e)
    Convert an UFL expression to a new UFL expression. All nonterminal object instances are replaced with identical copies, while terminal objects are kept. This is used for testing that objects in the expression behave as expected.

ufl.algorithms.traversal module

This module contains algorithms for traversing expression trees in different ways.

ufl.algorithms.traversal.iter_expressions(a)
    Utility function to handle Form, Integral and any Expr the same way when inspecting expressions. Returns an iterable over Expr instances: - a is an Expr: (a,) - a is an Integral: the integrand expression of a - a is a Form: all integrand expressions of all integrals

Module contents

This module collects algorithms and utility functions operating on UFL objects.

ufl.core package

Submodules

ufl.core.compute_expr_hash module

Non-recursive traversal based hash computation algorithm.
Fast iteration over nodes in an Expr DAG to compute memoized hashes for all unique nodes.

ufl.core.compute_expr_hash.compute_expr_hash(expr)

    Compute hashes of expr and all its nodes efficiently without using Python recursion.
ufl.core.expr module

This module defines the Expr class, the superclass for all expression tree node types in UFL.

NB! A note about other operators not implemented here:

More operators (special functions) on Exprs are defined in exprooperators.py, as well as the transpose “A.T” and spatial derivative “a.dx(i)”. This is to avoid circular dependencies between Expr and its subclasses.

```python
class ufl.core.expr.Expr
    Bases: object

Base class for all UFL expression types.

Instance properties
Every expression instance will have certain properties. Most important are the ufl_operands, ufl_shape, ufl_free_indices, and ufl_index_dimensions properties. Expressions are immutable and hashable.

Type traits
The Expr API defines a number of type traits that each subclass needs to provide. Most of these are specified indirectly via the arguments to the ufl_type class decorator, allowing UFL to do some consistency checks and automate most of the traits for most types. The type traits are accessed via a class or instance object on the form obj._ufl_traitname_. See the source code for description of each type trait.

Operators
Some Python special functions are implemented in this class, some are implemented in subclasses, and some are attached to this class in the ufl_type class decorator.

Defining subclasses
To define a new expression class, inherit from either Terminal or Operator, and apply the ufl_type class decorator with suitable arguments. See the docstring of ufl_type for details on its arguments. Looking at existing classes similar to the one you wish to add is a good idea. Looking through the comments in the Expr class and ufl_type to understand all the properties that may need to be specified is also a good idea. Note that many algorithms in UFL and form compilers will need handlers implemented for each new type.

```python
@ufl_type()
class MyOperator(Operator):
    pass
```

Type collections
All Expr subclasses are collected by ufl_type in global variables available via Expr.

Profiling
Object creation statistics can be collected by doing

```python
Expr.ufl_enable_profiling()
# ... run some code
initstats, delstats = Expr.ufl_disable_profiling()
```

Giving a list of creation and deletion counts for each typecode.

T
Transposed a rank two tensor expression. For more general transpose operations of higher order tensor expressions, use indexing and Tensor.

```python
cell()
domain()
domains()
dx(*ii)
```

Return the partial derivative with respect to spatial variable number i.

```python
evaluate(x, mapping, component, index_values)
```
Evaluate expression at given coordinate with given values for terminals.
free_indices()

generic_dimenson()
    Return the geometric dimension this expression lives in.

index_dimensions()

is_cellwise_constant()
    Return whether this expression is spatially constant over each cell.

operands()

rank()
    Return the tensor rank of the expression.

reconstruct (*operands)
    Return a new object of the same type with new operands.

shape()
    Return the tensor shape of the expression.

static ufl_disable_profiling()
    Turn off object counting mechanism. Returns object init and del counts.

ufl_domain()
    Return the single unique domain this expression is defined on or throw an error.

ufl_domains()
    Return all domains this expression is defined on.

static ufl_enable_profiling()
    Turn on object counting mechanism and reset counts to zero.

ufl.core.multiindex module

This module defines the single index types and some internal index utilities.

class ufl.core.multiindex.FixedIndex(value)
    Bases: ufl.core.multiindex.IndexBase
    UFL value: An index with a specific value assigned.

class ufl.core.multiindex.Index(count=None)
    Bases: ufl.core.multiindex.IndexBase
    UFL value: An index with no value assigned.
    Used to represent free indices in Einstein indexing notation.

    count()

class ufl.core.multiindex.IndexBase
    Bases: object

class ufl.core.multiindex.MultiIndex(indices)
    Bases: ufl.core.terminal.Terminal
    Represents a sequence of indices, either fixed or free.

    evaluate (x, mapping, component, index_values)

    indices()

    is_cellwise_constant()
ufl_domains()
    Return tuple of domains related to this terminal object.

ufl_free_indices
ufl_index_dimensions
ufl_shape

ufl.core.multiindex.as_multi_index(ii, shape=None)
ufl.core.multiindex.indices(n)
    UFL value: Return a tuple of n new Index objects.

ufl.core.operator module

Base class for all operators, i.e. non-terminal expr types.

class ufl.core.operator.Operator(operands=None)
    Bases: ufl.core.expr.Expr

    ufl_operands

ufl.core.terminal module

This module defines the Terminal class, the superclass for all types that are terminal nodes in the expression trees.

class ufl.core.terminal.FormArgument
    Bases: ufl.core.terminal.Terminal

class ufl.core.terminal.Terminal
    Bases: ufl.core.expr.Expr

    A terminal node in the UFL expression tree.

evaluate(x, mapping, component, index_values, derivatives=())
    Get self from mapping and return the component asked for.

    ufl_domains()
    Return tuple of domains related to this terminal object.

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_operands = ()

ufl.core.ufl_id module

Utilities for types with a globally counted unique id attached to each object.

ufl.core.ufl_id.attach_ufl_id(cls)
    Equip class with .ufl_id() and handle bookkeeping.

    Usage:

    # 1) Apply to class @attach_ufl_id class MyClass(object):
    # 2) If __slots__ is defined, include “_ufl_id” attribute __slots__ = (“_ufl_id”), # 3) Add keyword argument to constructor def __init__(self, *args, ufl_id=No...
# 4) Call self._init_ufl_id with ufl_id and assign to ._ufl_id attribute
self._init_ufl_id(ufl_id)

Result:
MyClass().ufl_id() returns unique value for each constructed object.

ufl.core.ufl_type module

ufl.core.ufl_type.attach_implementations_of_indexing_interface(cls, inherit_shape_from_operand, inherit_indices_from_operand)

ufl.core.ufl_type.attach_operators_from_hash_data(cls)
    Class decorator to attach __hash__, __eq__ and __ne__ implementations.
    These are implemented in terms of a .ufl_hash_data() method on the class, which should return a tuple or
    hashable and comparable data.

ufl.core.ufl_type.check_abstract_trait_consistency(cls)
    Check that the first base classes up to Expr are other UFL types.

ufl.core.ufl_type.check_has_slots(cls)
    Check if type has __slots__ unless it is marked as exception with _ufl_noslots_

ufl.core.ufl_type.check_implements_required_methods(cls)
    Check if type implements the required methods.

ufl.core.ufl_type.check_implements_required_properties(cls)
    Check if type implements the required properties.

ufl.core.ufl_type.check_is_terminal_consistency(cls)
    Check for consistency in is_terminal trait among superclasses.

ufl.core.ufl_type.check_type_traits_consistency(cls)
    Execute a variety of consistency checks on the ufl type traits.

ufl.core.ufl_type.determine_num_ops(cls, num_ops, unop, binop, rbinop)

ufl.core.ufl_type.get_base_attr(cls, name)
    Return first non-None attribute of given name among base classes.

ufl.core.ufl_type.set_trait(cls, basename, value, inherit=False)
    Assign a trait to class with namespaces _ufl_basename_ applied.
    If trait value is None, optionally inherit it from the closest base class that has it.

ufl.core.ufl_type.ufl_type(is_abstract=False, is_terminal=None, is_scalar=False, is_index_free=False, is_shaping=False, is_literal=False, is_terminal_modifier=False, is_in_reference_frame=False, is_restriction=False, is_evaluation=False, is_differential=None, use_default_hash=True, num_ops=None, inherit_shape_from_operand=None, inherit_indices_from_operand=None, wraps_type=None, unop=None, binop=None, rbinop=None)

This decorator is to be applied to every subclass in the UFL Expr hierarchy.

This decorator contains a number of checks that are intended to enforce uniform behaviour across UFL types.

The rationale behind the checks and the meaning of the optional arguments should be sufficiently documented
in the source code below.
**ufl.core.ufl_type.update_global_expr_attributes(cls)**

Update global Expr attributes, mainly by adding cls to global collections of ufl types.

**Module contents**

**ufl.corealg package**

**Submodules**

**ufl.corealg.map_dag module**

Basic algorithms for applying functions to subexpressions.

**ufl.corealg.map_dag.map_expr_dag(function, expression, compress=True)**

Apply a function to each subexpression node in expression dag.

- If compress is True (default), the output object from the function is cached in a dict and reused such that the resulting expression dag does not contain duplicate objects.
- Returns the result of the final function call.

**ufl.corealg.multifunction module**

Base class for multifunctions with UFL Expr type dispatch.

**class ufl.corealg.multifunction.MultiFunction**

Bases: object

Base class for collections of nonrecursive expression node handlers.

- Subclass this (remember to call the __init__ method of this class), and implement handler functions for each Expr type, using the lower case handler name of the type (exprtype._ufl_handler_name_).

This class is optimized for efficient type based dispatch in the __call__ operator via typecode based lookup of the handler function bound to the algorithm object. Of course function call overhead of Python still applies.

**expr(o, *args)**

Trigger error for types with missing handlers.

**reuse_if_untouched(o, *ops)**

Reuse object if operands are the same objects.

- Use in your own subclass by setting e.g.
  
  ```python
  expr = MultiFunction.reuse_if_untouched
  ```

  as a default rule.

**undefined(o, *args)**

Trigger error for types with missing handlers.

**ufl.corealg.multifunction.get_num_args(function)**

**ufl.corealg.multifunction.memoized_handler(handler)**

Function decorator to memoize MultiFunction handlers.
ufl.corealg.traversal module

Various expression traversal utilities.

The algorithms here are non-recursive, which is faster than recursion by a factor 10 or so because of the function call overhead.

- **cutoff_post_traversal**(expr, cutofftypes)
  Yields o for each node o in expr, child before parent, but skipping subtrees of the cutofftypes.

- **post_traversal**(expr)
  Yields o for each node o in expr, child before parent.

- **pre_traversal**(expr)
  Yields o for each tree node o in expr, parent before child.

- **traverse_terminals**(expr)
  Iterate over all terminal objects in expression, including duplicates.

- **traverse_unique_terminals**(expr)
  Iterate over all terminal objects in expression, not including duplicates.

- **unique_post_traversal**(expr, visited=None)
  Yields o for each node o in expr, child before parent.
  Never visits a node twice.

- **unique_pre_traversal**(expr, visited=None)
  Yields o for each tree node o in expr, parent before child.
  This version only visits each node once!

Module contents

ufl.finiteelement package

Submodules

- **broken element module**

  class **BrokenElement**(element)
  
  Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

  The discontinuous version of an existing Finite Element space

  - **mapping**()
  - **shortstr**()

- **element list module**

  This module provides an extensive list of predefined finite element families. Users or more likely, form compilers, may register new elements by calling the function register_element.

  **canonical_element_description**(family, cell, order, form_degree)

  Given basic element information, return corresponding element information on canonical form.
Input: family, cell, (polynomial) order, form_degree
Output: family (canonical), short_name (for printing), order, value shape, reference value shape, sobolev_space

This is used by the FiniteElement constructor to validate input data against the element list and aliases defined in ufl.

```python
ufl.finiteelement.elementlist.feec_element(family, n, r, k)
ufl.finiteelement.elementlist.register_alias(alias, to)
ufl.finiteelement.elementlist.register_element(family, short_name, value_rank, sobolev_space, mapping, degree_range, cellnames)
```

Register new finite element family

```python
ufl.finiteelement.elementlist.register_element2(family, value_rank, sobolev_space, mapping, degree_range, cellnames)
```

Register new finite element family

```python
ufl.finiteelement.elementlist.show_elements()
```

**ufl.finiteelement.enrichedelement module**

This module defines the UFL finite element classes.

```python
class ufl.finiteelement.enrichedelement.EnrichedElement(*elements)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
```

The vector sum of two finite element spaces:

EnrichedElement(V, Q) = \{v + q | v in V, q in Q\}.

```python
is_cellwise_constant()
```

Return whether the basis functions of this element is spatially constant over each cell.

```python
mapping()
shortstr()
```

Format as string for pretty printing.

**ufl.finiteelement.facetelement module**

```python
class ufl.finiteelement.facetelement.FacetElement(element)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
```

A version of an existing Finite Element space in which all dofs associated with the interior have been discarded

```python
mapping()
shortstr()
```

**ufl.finiteelement.finiteelement module**

This module defines the UFL finite element classes.

```python
class ufl.finiteelement.finiteelement.FiniteElement(family, cell=None, degree=None, form_degree=None, quad_scheme=None)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
```

The basic finite element class for all simple finite elements
mapping()

shortstr()
    Format as string for pretty printing.

sobolev_space()

**ufl.finiteelement.finiteelementbase module**

This module defines the UFL finite element classes.

```python
class ufl.finiteelement.finiteelementbase.FiniteElementBase(family, cell, degree, quad_scheme, value_shape, reference_value_shape)
```

**Bases: object**

Base class for all finite elements

```python
cell()
    Return cell of finite element

degree([component=None])
    Return polynomial degree of finite element

extract_component(i)
    Recursively extract component index relative to a (simple) element and that element for given value component index

extract_reference_component(i)
    Recursively extract reference component index relative to a (simple) element and that element for given reference value component index

extract_subelement_component(i)
    Extract direct subelement index and subelement relative component index for a given component index

extract_subelement_reference_component(i)
    Extract direct subelement index and subelement relative reference component index for a given reference component index

family()
    Return finite element family

is_cellwise_constant([component=None])
    Return whether the basis functions of this element is spatially constant over each cell.

mapping()

num_sub_elements()
    Return number of sub elements

quadrature_scheme()
    Return quadrature scheme of finite element

reference_value_shape()
    Return the shape of the value space on the reference cell.

reference_value_size()
    Return the integer product of the reference value shape.

sub_elements()
    Return list of sub elements
symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

value_shape()
    Return the shape of the value space on the global domain.

value_size()
    Return the integer product of the value shape.

ufl.finiteelement.hdivcurl module

class ufl.finiteelement.hdivcurl.HCurlElement(element)
    Bases: ufl.finiteelement.outerproductelement.OuterProductElement
    A curl-conforming version of an outer product element, assuming this makes mathematical sense.
    shortstr()

class ufl.finiteelement.hdivcurl.HDivElement(element)
    Bases: ufl.finiteelement.outerproductelement.OuterProductElement
    A div-conforming version of an outer product element, assuming this makes mathematical sense.
    shortstr()

ufl.finiteelement.interiorelement module

class ufl.finiteelement.interiorelement.InteriorElement(element)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
    A version of an existing Finite Element space in which only the dofs associated with the interior have been kept
    mapping()
    shortstr()

ufl.finiteelement.mixedelement module

This module defines the UFL finite element classes.

class ufl.finiteelement.mixedelement.MixedElement(*elements, **kwargs)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
    A finite element composed of a nested hierarchy of mixed or simple elements
    degree(component=None)
        Return polynomial degree of finite element
    extract_component(i)
        Recursively extract component index relative to a (simple) element and that element for given value component index
    extract_reference_component(i)
        Recursively extract reference_component index relative to a (simple) element and that element for given value reference_component index
extract_subelement_component \( (i) \)
Extract direct subelement index and subelement relative component index for a given component index.

extract_subelement_reference_component \( (i) \)
Extract direct subelement index and subelement relative reference_component index for a given reference_component index.

is_cellwise_constant \( (\text{component} = \text{None}) \)
Return whether the basis functions of this element is spatially constant over each cell.

mapping()
num_sub_elements()
Return number of sub elements.

reconstruct_from_elements \( (*elements) \)
Reconstruct a mixed element from new subelements.

shortstr()
Format as string for pretty printing.

sub_elements()
Return list of sub elements.

symmetry()
Return the symmetry dict, which is a mapping \( c_0 \) -> \( c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.finiteelement.mixedelement.TensorElement (\text{family}, \text{cell}, \text{degree}, \text{shape} = \text{None}, \text{symmetry} = \text{None}, \text{quad_scheme} = \text{None})
Bases: ufl.finiteelement.mixedelement.MixedElement

A special case of a mixed finite element where all elements are equal.

extract_subelement_component \( (i) \)
Extract direct subelement index and subelement relative component index for a given component index.

flattened_sub_element_mapping()
mapping()
shortstr()
Format as string for pretty printing.

symmetry()
Return the symmetry dict, which is a mapping \( c_0 \) -> \( c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \).

class ufl.finiteelement.mixedelement.VectorElement (\text{family}, \text{cell}, \text{degree}, \text{dim} = \text{None}, \text{form_degree} = \text{None}, \text{quad_scheme} = \text{None})
Bases: ufl.finiteelement.mixedelement.MixedElement

A special case of a mixed finite element where all elements are equal.

shortstr()
Format as string for pretty printing.

ufl.finiteelement.outerproductelement module

This module defines the UFL finite element classes.

2.2. ufl package
class `ufl.finiteelement.outerproductelement.OuterProductElement(A, B, cell=None)`

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

The outer (tensor) product of 2 element spaces:

\[ V = A \otimes B \]

Given bases \( \phi_A, \phi_B \) for \( A, B \), \( \phi_A \ast \phi_B \) forms a basis for \( V \).

mapping()  
shortstr()  
Short pretty-print.

class `ufl.finiteelement.outerproductelement.OuterProductVectorElement(A, B, cell=None, dim=None)`

Bases: `ufl.finiteelement.mixedelement.MixedElement`

A special case of a mixed finite element where all elements are equal OuterProductElements

mapping()  
shortstr()  
Format as string for pretty printing.

`ufl.finiteelement.restrictedelement` module

This module defines the UFL finite element classes.

class `ufl.finiteelement.restrictedelement.RestrictedElement(element, restriction_domain)`

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

Represents the restriction of a finite element to a type of cell entity.

element()  
is_cellwise_constant()  
Return whether the basis functions of this element is spatially constant over each cell.

mapping()  
num_restricted_sub_elements()  
Return number of restricted sub elements.

num_sub_elements()  
Return number of sub elements

restricted_sub_elements()  
Return list of restricted sub elements.

restriction_domain()  
Return the domain onto which the element is restricted.

shortstr()  
Format as string for pretty printing.

sub_element()  
Return the element which is restricted.

sub_elements()  
Return list of sub elements
symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

**ufl.finiteelement.tensorproductelement module**

This module defines the UFL finite element classes.

class ufl.finiteelement.tensorproductelement.TensorProductElement(elements)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    The tensor product of d element spaces:
    \[ V = V_0 \otimes V_1 \otimes \ldots \otimes V_d \]

    Given bases \{\phi_i\} for \(V_i\) for \(i = 1, \ldots, d\), \{phi_0 \ast phi_1 \ast \ldots \ast phi_d\} forms a basis for \(V\).

    mapping()
    num_sub_elements()
    shortstr()
    sub_elements()

**ufl.finiteelement.traceelement module**

class ufl.finiteelement.traceelement.TraceElement(element)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    A finite element space: the trace of a given hdiv element. This is effectively a scalar-valued restriction which is non-zero only on cell facets.

    mapping()
    shortstr()

**Module contents**

This module defines the UFL finite element classes.

**ufl.formatting package**

**Submodules**

**ufl.formatting.graph module**

Algorithms for working with linearized computational graphs.

class ufl.formatting.graph.Graph(expression)
    Graph class which computes connectivity on demand.

E()
class `ufl.formatting.graph.HeapItem` *(incoming, outgoing, i)*

Bases: `object`

Given an expr, returns a frozenset of its dependencies.

**Possible dependency values are:**
- “c” - depends on runtime information like the cell, local<>global coordinate mappings, facet normals, or coefficients
- “x” - depends on local coordinates
- “v%d” % i - depends on argument i, for i in [0, rank)

**argument** *(x)*

**coefficient** *(x)*

**expr** *(o)*

**facet_normal** *(o)*

**geometric_quantity** *(x)*

**spatial_derivative** *(o)*

```python
ufl.formatting.graph.build_graph(expr)
```

Build a linearized graph from an UFL Expr.

Returns $G = (V, E)$, with $V$ being a list of graph nodes (Expr objects) in post traversal ordering and $E$ being a list of edges. Each edge is represented as a $(i, j)$ tuple where $i$ and $j$ are vertex indices into $V$.

```python
ufl.formatting.graph.depth_first_ordering(G)
```

```python
ufl.formatting.graph.extract_incoming_edges(G)
```

Build lists of incoming edges to each vertex in a linearized graph.

```python
ufl.formatting.graph.extract_incoming_vertex_connections(G)
```

Build lists of vertices in incoming and outgoing edges to and from each vertex in a linearized graph.

Returns lists Vin and Vout.

```python
ufl.formatting.graph.extract_outgoing_edges(G)
```

Build list of outgoing edges from each vertex in a linearized graph.

```python
ufl.formatting.graph.extract_outgoing_vertex_connections(G)
```

Build lists of vertices in incoming and outgoing edges to and from each vertex in a linearized graph.

Returns lists Vin and Vout.

```python
ufl.formatting.graph.len_items(sequence)
```

```python
ufl.formatting.graph.lists(n)
```

```python
ufl.formatting.graph.partition(G, criteria=string_set_criteria)
```

```python
ufl.formatting.graph.string_set_criteria(v, keys)
```
ufl.formatting.latextools module

This module defines basic utilities for stitching together LaTeX documents.

- `ufl.formatting.latextools.align(lines)`
- `ufl.formatting.latextools.document(title, sections)`
- `ufl.formatting.latextools.itemize(items)`
- `ufl.formatting.latextools.section(s)`
- `ufl.formatting.latextools.subsection(s)`
- `ufl.formatting.latextools.subsubsection(s)`
- `ufl.formatting.latextools.testdocument()`
- `ufl.formatting.latextools.verbatim(string)`

ufl.formatting.printing module

A collection of utility algorithms for printing of UFL objects, mostly intended for debugging purposes.

- `ufl.formatting.printing.form_info(form)`
- `ufl.formatting.printing.integral_info(integral)`
- `ufl.formatting.printing.tree_format(expression, indentation=0, parentheses=True)`

ufl.formatting.ufl2dot module

A collection of utility algorithms for printing of UFL objects in the DOT graph visualization language, mostly intended for debugging purposes.

```python
class ufl.formatting.ufl2dot.CompactLabeller(function_mapping=None):
    Bases: ufl.formatting.ufl2dot.ReprLabeller
    cell_avg(e)
    component_tensor(e)
    curl(e)
    determinant(e)
    dev(e)
    diff(e)
    div(e)
    division(e)
    dot(e)
    facet_avg(e)
    form_argument(e)
    geometric_quantity(e)
    grad(e)
    identity(e)
```
\texttt{index\_sum}(e)
\texttt{indexed}(e)
\texttt{inner}(e)
\texttt{math\_function}(e)
\texttt{multi\_index}(e)
\texttt{nabla\_div}(e)
\texttt{nabla\_grad}(e)
\texttt{negative\_restricted}(e)
\texttt{outer}(e)
\texttt{positive\_restricted}(e)
\texttt{power}(e)
\texttt{product}(e)
\texttt{scalar\_value}(e)
\texttt{skew}(e)
\texttt{sum}(e)
\texttt{trace}(e)
\texttt{transposed}(e)
\texttt{zero}(e)

\textbf{class} \texttt{ufl.formatting.ufl2dot.FancyLabeller}(\texttt{function\_mapping=None})
\texttt{Bases:} \texttt{ufl.formatting.ufl2dot.CompactLabeller}

\textbf{class} \texttt{ufl.formatting.ufl2dot.ReprLabeller}
\texttt{Bases:} \texttt{ufl.corealg.multifunction.MultiFunction}
\texttt{operator}(e)
\texttt{terminal}(e)

\texttt{ufl.formatting.ufl2dot.build\_entities}(e, \texttt{nodes}, \texttt{edges}, \texttt{nodeoffset}, \texttt{prefix=''}, \texttt{labeller=None})
\texttt{ufl.formatting.ufl2dot.format\_entities}(\texttt{nodes}, \texttt{edges})
\texttt{ufl.formatting.ufl2dot.ufl2dot}(\texttt{expression}, \texttt{formname='a'}, \texttt{nodeoffset=0}, \texttt{begin=True}, \texttt{end=True}, \texttt{labeling='repr'}, \texttt{object\_names=None})

\texttt{ufl.formatting.ufl2latex module}

This module defines expression transformation utilities, either converting UFL expressions to new UFL expressions or converting UFL expressions to other representations.

\textbf{class} \texttt{ufl.formatting.ufl2latex.Expression2LatexHandler}(\texttt{argument\_names=None}, \texttt{coefficient\_names=None})
\texttt{Bases:} \texttt{ufl.algorithms.transformer.Transformer}
\texttt{abs}(o, a)
\texttt{acos}(o, f)
\texttt{and\_condition}(o, a, b)
argument\( (o) \)
asin\( (o,f) \)
atan\( (o,f) \)
atan2\( (o,f1,f2) \)
bessel\_K\( (o,nu,f) \)
bessel\_i\( (o,nu,f) \)
bessel\_j\( (o,nu,f) \)
bessel\_y\( (o,nu,f) \)
cell\_avg\( (o,f) \)
coefficient\( (o) \)
coefficient\_derivative\( (o,f,w,v) \)
cofactor\( (o,A) \)
component\_tensor\( (o, *ops) \)
conditional\( (o, c, t, f) \)
constant\( (o) \)
\cos\( (o,f) \)
\cosh\( (o,f) \)
cross\( (o,a,b) \)
curl\( (o,f) \)
determinant\( (o,A) \)
deviatoric\( (o,A) \)
div\( (o,f) \)
division\( (o,a,b) \)
dot\( (o,a,b) \)
eq\( (o,a,b) \)
erf\( (o,f) \)
exp\( (o,f) \)
expr\( (o) \)
facet\_normal\( (o) \)
ge\( (o,a,b) \)
grad\( (o,f) \)
gt\( (o,a,b) \)
identity\( (o) \)
index\_sum\( (o,f,i) \)
indexed\( (o,a,b) \)
inner\( (o,a,b) \)
inverse(o, A)
le(o, a, b)
list_tensor(o)
ln(o, f)
lt(o, a, b)
max_value(o, a, b)
min_value(o, a, b)
multi_index(o)
nabla_div(o, f)
nabla_grad(o, f)
ne(o, a, b)
negative_restricted(o, f)
not_condition(o, a)
or_condition(o, a, b)
outer(o, a, b)
permutation_symbol(o)
positive_restricted(o, f)
power(o, a, b)
product(o, *ops)
scalar_value(o)
sin(o, f)
sinh(o, f)
skew(o, A)
sqrt(o, f)
sum(o, *ops)
sym(o, A)
tan(o, f)
tanh(o, f)
trace(o, A)
transposed(o, a)
variable(o)
variable_derivative(o, f, v)
zero(o)

ufl.formatting.ufl2latex.bfname(i, p)
ufl.formatting.ufl2latex.build_precedence_map()
ufl.formatting.ufl2latex.cfname(i)
ufl.formatting.ufl2latex.code2latex(G, partitions, formdata)
    TODO: Document me
ufl.formatting.ufl2latex.dependency_sorting(deplist, rank)
ufl.formatting.ufl2latex.deps2latex(deps)
ufl.formatting.ufl2latex.element2latex(element)
ufl.formatting.ufl2latex.expression2latex(expression, argument_names=None, coefficient_names=None)
ufl.formatting.ufl2latex.form2code2latex(formdata)
ufl.formatting.ufl2latex.form2latex(form, formdata)
ufl.formatting.ufl2latex.format_index(ii)
ufl.formatting.ufl2latex.format_multi_index(ii, formatstring='%s')
ufl.formatting.ufl2latex.formdata2latex(formdata)
    Render forms from a .ufl file as a LaTeX document.
ufl.formatting.ufl2latex.integrand2code(integrand, formdata)
ufl.formatting.ufl2latex.par(s, condition=True)
ufl.formatting.ufl2latex.tex2pdf(latexfilename, pdffilename)
ufl.formatting.ufl2latex.ufl2latex(expression)
    Generate LaTeX code for a UFL expression or form (wrapper for form2latex and expression2latex).
ufl.formatting.ufl2latex.ufl2pdf(uflfilename, latexfilename, pdffilename, compile=False)
ufl.formatting.ufl2latex.ufl2tex(uflfilename, latexfilename, compile=False)
    Compile a .tex file from a .ufl file.

Module contents

ufl.utils package

Submodules

ufl.utils.counted module

Utilities for types with a global unique counter attached to each object.

class ufl.utils.counted.ExampleCounted(count=None)
    Bases: object
    An example class for classes of objects identified by a global counter.
    Mimic this class to create globally counted objects within a single type.
    count()

ufl.utils.counted.counted_init(self, count=None, countedclass=None)
    Initialize a counted object, see ExampleCounted below for how to use.
ufl.utils.derivativetuples module

This module contains a collection of utilities for representing partial derivatives as integer tuples.

ufl.utils.derivativetuples.\texttt{compute\_derivative\_tuples} \((n, gdim)\)

Compute the list of all derivative tuples for derivatives of given total order \(n\) and given geometric dimension \(gdim\). This function returns two lists. The first is a list of tuples, where each tuple of length \(n\) specifies the coordinate directions of the \(n\) derivatives. The second is a corresponding list of tuples, where each tuple of length \(gdim\) specifies the number of derivatives in each direction. Both lists have length \(gdim^n\) and are ordered as expected by the UFC function \texttt{tabulate\_basis\_derivatives}.

Example: If \(n = 2\) and \(gdim = 3\), then the nice tuples are

\[
(0, 0) \leftrightarrow (2, 0, 0) \leftrightarrow d^2/dxdx \leftrightarrow (0, 1) \leftrightarrow (1, 1, 0) \leftrightarrow d^2/dxdy \leftrightarrow (0, 2) \leftrightarrow (1, 0, 1) \leftrightarrow d^2/dxdz
\]
\[
(1, 0) \leftrightarrow (1, 1, 0) \leftrightarrow d^2/dydx \leftrightarrow (0, 2) \leftrightarrow (0, 1, 1) \leftrightarrow d^2/dydy \leftrightarrow (0, 1, 1) \leftrightarrow (0, 0, 2) \leftrightarrow d^2/dydz
\]
\[
(2, 0) \leftrightarrow (1, 0, 1) \leftrightarrow d^2/dzdx \leftrightarrow (2, 1) \leftrightarrow (0, 1, 1) \leftrightarrow d^2/dzdy \leftrightarrow (2, 2) \leftrightarrow (0, 0, 2) \leftrightarrow d^2/dzdz
\]

ufl.utils.derivativetuples.\texttt{derivative\_counts\_to\_listing} \((\text{derivative\_counts})\)

Convert a derivative count tuple to a derivative listing tuple.

The derivative \(d^3 / dy^2 dz\) is represented in counting form as \((0, 2, 1)\) meaning \((dx^0, dy^2, dz^1)\) and in listing form as \((1, 1, 2)\) meaning \((dy, dy, dz)\).

ufl.utils.derivativetuples.\texttt{derivative\_listing\_to\_counts} \((\text{derivatives}, gdim)\)

Convert a derivative listing tuple to a derivative count tuple.

The derivative \(d^3 / dy^2 dz\) is represented in counting form as \((0, 2, 1)\) meaning \((dx^0, dy^2, dz^1)\) and in listing form as \((1, 1, 2)\) meaning \((dy, dy, dz)\).

ufl.utils.dicts module

Various dict manipulation utilities.

\texttt{class ufl.utils.dicts.EmptyDictType}

Bases: dict

\texttt{update(*args, **kwargs)}

ufl.utils.dicts.\texttt{dict\_sum} \((\text{items})\)

Construct a dict, in between dict(items) and sum(items), by accumulating items for each key.

ufl.utils.dicts.\texttt{mergedicts} \((\text{dicts})\)

ufl.utils.dicts.\texttt{mergedicts2} \((d1, d2)\)

ufl.utils.dicts.\texttt{slice\_dict} \((\text{dictionary, keys, default=None})\)

ufl.utils.dicts.\texttt{some\_key} \((a\_dict)\)

Return an arbitrary key from a dictionary.

ufl.utils.dicts.\texttt{split\_dict} \((d, criteria)\)

Split a dict \(d\) into two dicts based on a criteria on the keys.

ufl.utils.dicts.\texttt{subdict} \((\text{superdict, keys})\)

ufl.utils.formatting module

Various string formatting utilities.
ufl.utils.formatting.**camel2underscore**(name)
    Convert a CamelCaps string to underscore_syntax.

ufl.utils.formatting.**dstr**(d, colsize=80)
    Pretty-print dictionary of key-value pairs.

ufl.utils.formatting.**estr**(elements)
    Format list of elements for printing.

ufl.utils.formatting.**istr**(o)
    Format object as string, inserting ? for None.

ufl.utils.formatting.**lstr**(l)
    Pretty-print list or tuple, invoking str() on items instead of repr() like str() does.

ufl.utils.formatting.**sstr**(s)
    Pretty-print set.

ufl.utils.formatting.**tstr**(t, colsize=80)
    Pretty-print list of tuples of key-value pairs.

**ufl.utils.indexflattening module**

This module contains a collection of utilities for mapping between multiindices and a flattened index space.

ufl.utils.indexflattening.**flatten_multiindex**(ii, strides)
    Return the flat index corresponding to the given multiindex.

ufl.utils.indexflattening.**shape_to_strides**(sh)
    Return a tuple of strides given a shape tuple.

ufl.utils.indexflattening.**unflatten_index**(i, strides)
    Return the multiindex corresponding to the given flat index.

**ufl.utils.sequences module**

Various sequence manipulation utilities.

ufl.utils.sequences.**and_tuples**(seqa, seqb)
    Return ‘and’ of all pairs in two sequences of same length.

ufl.utils.sequences.**iter_tree**(tree)
    Iterate over all nodes in a tree represented by lists of lists of leaves.

ufl.utils.sequences.**or_tuples**(seqa, seqb)
    Return ‘or’ of all pairs in two sequences of same length.

ufl.utils.sequences.**product**(sequence)
    Return the product of all elements in a sequence.

ufl.utils.sequences.**recursive_chain**(lists)

ufl.utils.sequences.**unzip**(seq)
    Inverse operation of zip: unzip(zip(a, b)) == (a, b)

ufl.utils.sequences.**xor**(a, b)
**ufl.utils.sorting module**

Utilities for sorting.

- **ufl.utils.sorting.canonicalize_metadata(metadata)**
  - Transform dict to a tuple of (key, value) item tuples ordered by key, with dict, list and tuple values converted the same way recursively. Lists and tuples are converted to tuples. Other values are converted using str(). This is such that the end result can be hashed and sorted using regular <, because python 3 doesn't allow e.g. (3 < "auto") which occurs regularly in metadata.

- **ufl.utils.sorting.sorted_by_count(seq)**
  - Sort a sequence by the item.count().

- **ufl.utils.sorting.sorted_by_key(mapping)**
  - Sort dict items by key, allowing different key types.

- **ufl.utils.sorting.sorted_by_ufl_id(seq)**
  - Sort a sequence by the item.ufl_id().

- **ufl.utils.sorting.topological_sorting(nodes, edges)**
  - Return a topologically sorted list of the nodes
  - Implemented algorithm from Wikipedia :P
  - No error for cyclic edges...

**ufl.utils.stacks module**

Various utility data structures.

- **class ufl.utils.stacks.Stack(*args)**
  - Bases: list
  - A stack datastructure.
    - `peek()`
    - `push(v)`

- **class ufl.utils.stacks.StackDict(*args, **kwargs)**
  - Bases: dict
  - A dict that can be changed incrementally with `d.push(k,v)` and have changes rolled back with `k,v = d.pop()`.
    - `pop()`
    - `push(k, v)`

**ufl.utils.system module**

Various utilities accessing system io.

- **ufl.utils.system.get_status_output(cmd, input=None, cwd=None, env=None)**
- **ufl.utils.system.openpdf(pdffilename)**
  - Open PDF file in external pdf viewer.
ufl.utils.system.pdflatex \textit{(latexfilename, pdffilename, flags='')} 
Execute pdflatex to compile a latex file into pdf.

ufl.utils.system.write_file \textit{(filename, text)}

\textbf{ufl.utils.timer module}

Timer utilities.

\texttt{class ufl.utils.timer.Timer(name)}

\texttt{Bases: object}

\texttt{end()}

\textbf{ufl.utils.ufltypedicts module}

Various utility data structures.

\texttt{class ufl.utils.ufltypedicts.UFLTypeDefaultDict(default)}

\texttt{Bases: dict}

\texttt{class ufl.utils.ufltypedicts.UFLTypeDict}

\texttt{Bases: dict}

\textbf{Module contents}

\section{2.2.2 Submodules}

\section{2.2.3 ufl.algebra module}

Basic algebra operations.

\texttt{class ufl.algebra.Abs(a)}

\texttt{Bases: ufl.core.operator.Operator}

\texttt{evaluate(x, mapping, component, index_values)}

\texttt{ufl_free_indices}

\texttt{ufl_index_dimensions}

\texttt{ufl_shape}

\texttt{class ufl.algebra.Division(a, b)}

\texttt{Bases: ufl.core.operator.Operator}

\texttt{evaluate(x, mapping, component, index_values)}

\texttt{ufl_free_indices}

\texttt{ufl_index_dimensions}

\texttt{ufl_shape = ()}

\texttt{class ufl.algebra.Power(a, b)}

\texttt{Bases: ufl.core.operator.Operator}

\texttt{evaluate(x, mapping, component, index_values)}

\texttt{ufl_free_indices}
```
import ufl

class ufl_index_dimensions:
    ufl_shape = ()

class ufl.algebra.Product(a, b):
    Bases: ufl.core.operator.Operator
    The product of two or more UFL objects.
    evaluate(x, mapping, component, index_values)

class ufl_shape:
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.algebra.Sum(a, b):
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)

2.2.4 ufl.argument module

This module defines the class Argument and a number of related classes (functions), including TestFunction and TrialFunction.

class ufl.argument.Argument(function_space, number, part=None):
    Bases: ufl.core.terminal.FormArgument
    UFL value: Representation of an argument to a form.
    element()
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
    number()
    part()
    ufl_domain()
    ufl_domains()
        Return tuple of domains related to this terminal object.
    ufl_element()
    ufl_function_space()
        Get the function space of this Argument.
    ufl_shape

class ufl.argument.Arguments(function_space, number):
    UFL value: Create an Argument in a mixed space, and return a tuple with the function components corresponding to the subelements.

class ufl.argument.TestFunction(function_space, part=None):
    UFL value: Create a test function argument to a form.
```
ufl.argument.TestFunctions(function_space)
UFL value: Create a TestFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

ufl.argument.TrialFunction(function_space, part=None)
UFL value: Create a trial function argument to a form.

ufl.argument.TrialFunctions(function_space)
UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

2.2.5 ufl.assertions module

This module provides assertion functions used by the UFL implementation.

ufl.assertions.expecting_expr(v)
ufl.assertions.expecting_instance(v, c)
ufl.assertions.expecting_python_scalar(v)
ufl.assertions.expecting_terminal(v)
ufl.assertions.expecting_true_ufl_scalar(v)
ufl.assertions.ufl_assert(condition, *message)
Assert that condition is true and otherwise issue an error with given message.

2.2.6 ufl.cell module

Types for representing a cell.

class ufl.cell.AbstractCell(topological_dimension, geometric_dimension)
  Bases: object
  Representation of an abstract finite element cell with only the dimensions known.
  
generic_dimension()
  Return the dimension of the space this cell is embedded in.

has_simplex_facets()
  Return True if all the facets of this cell are simplex cells.

is_simplex()
  Return True if this is a simplex cell.

topological_dimension()
  Return the dimension of the topology of this cell.

class ufl.cell.Cell(cellname, geometric_dimension=None)
  Bases: ufl.cell.AbstractCell
  Representation of a named finite element cell with known structure.

  cellname()
    Return the cellname of the cell.

  has_simplex_facets()

  is_simplex()
num_edges()
   The number of cell edges.

num_facet_edges()
   The number of facet edges.

num_facets()
   The number of cell facets.

num_vertices()
   The number of cell vertices.

class ufl.cell.OuterProductCell(A, B, gdim=None)
   Bases: ufl.cell.AbstractCell
   Representation of a cell formed as the Cartesian product of two existing cells

   facet_horiz

   facet_vert

   has_simplex_facets()
      Return True if all the facets of this cell are simplex cells.

   is_simplex()
      Return True if this is a simplex cell.

   num_edges()
      The number of cell edges.

   num_facets()
      The number of cell facets.

   num_vertices()
      The number of cell vertices.

class ufl.cell.TensorProductCell(cells)
   Bases: ufl.cell.AbstractCell

   has_simplex_facets()
      Return True if all the facets of this cell are simplex cells.

   is_simplex()
      Return True if this is a simplex cell.

   num_edges()
      The number of cell edges.

   num_facets()
      The number of cell facets.

   num_vertices()
      The number of cell vertices.

   sub_cells()
      Return list of cell factors.

ufl.cell.as_cell(cell)
   Convert any valid object to a Cell or return cell if it is already a Cell.

   Allows an already valid cell, a known cellname string, or a tuple of cells for a product cell.

ufl.cell.hypercube(topological_dimension, geometric_dimension=None)
   Return a hypercube cell of given dimension.
ufl.cell.simplex(topological_dimension, geometric_dimension=None)
    Return a simplex cell of given dimension.

2.2.7 ufl.checks module

Utility functions for checking properties of expressions.

ufl.checks.is_cellwise_constant(expr)
    Return whether expression is constant over a single cell.

ufl.checks.is_globally_constant(expr)
    Check if an expression is globally constant, which includes spatially independent constant coefficients that are not known before assembly time.

ufl.checks.is_python_scalar(expression)
    Return True iff expression is of a Python scalar type.

ufl.checks.is_scalar_constant_expression(expr)
    Check if an expression is a globally constant scalar expression.

ufl.checks.is_true_ufl_scalar(expression)
    Return True iff expression is scalar-valued, with no free indices.

ufl.checks.is_ufl_scalar(expression)
    Return True iff expression is scalar-valued, but possibly containing free indices.

2.2.8 ufl.classes module

This file is useful for external code like tests and form compilers, since it enables the syntax “from ufl.classes import CellFacetoOBar” for getting implementation details not exposed through the default ufl namespace. It also contains functionality used by algorithms for dealing with groups of classes, and for mapping types to different handler functions.

class ufl.classes.Expr
    Bases: object

    Base class for all UFL expression types.

    Instance properties Every expression instance will have certain properties. Most important are the ufl_operands, ufl_shape, ufl_free_indices, and ufl_index_dimensions properties. Expressions are immutable and hashable.

    Type traits The Expr API defines a number of type traits that each subclass needs to provide. Most of these are specified indirectly via the arguments to the ufl_type class decorator, allowing UFL to do some consistency checks and automate most of the traits for most types. The type traits are accessed via a class or instance object on the form obj._ufl_traitname_. See the source code for description of each type trait.

    Operators Some Python special functions are implemented in this class, some are implemented in subclasses, and some are attached to this class in the ufl_type class decorator.

    Defining subclasses To define a new expression class, inherit from either Terminal or Operator, and apply the ufl_type class decorator with suitable arguments. See the docstring of ufl_type for details on its arguments. Looking at existing classes similar to the one you wish to add is a good idea. Looking through the comments in the Expr class and ufl_type to understand all the properties that may need to be specified is also a good idea. Note that many algorithms in UFL and form compilers will need handlers implemented for each new type.

2.2. ufl package
@ufl_type()
class MyOperator(Operator):
    pass

**Type collections**  All `Expr` subclasses are collected by `ufl_type` in global variables available via `Expr`.

**Profiling**  Object creation statistics can be collected by doing

```python
Expr.ufl_enable_profiling()
# ... run some code
initstats, delstats = Expr.ufl_disable_profiling()
```

Giving a list of creation and deletion counts for each typecode.

```python
T
```
Transposed a rank two tensor expression. For more general transpose operations of higher order tensor expressions, use indexing and Tensor.

```python
cell()
domain()
domains()
```

```python
dx(*ii)
```
Return the partial derivative with respect to spatial variable number i.

```python
evaluate(x, mapping, component, index_values)
```
Evaluate expression at given coordinate with given values for terminals.

```python
free_indices()
geometric_dimension()
```
Return the geometric dimension this expression lives in.

```python
index_dimensions()
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
operands()
rank()
```
Return the tensor rank of the expression.

```python
reconstruct(*operands)
```
Return a new object of the same type with new operands.

```python
shape()
```
Return the tensor shape of the expression.

```python
static ufl_disable_profiling()
```
Turn off object counting mechanism. Returns object init and del counts.

```python
ufl_domain()
```
Return the single unique domain this expression is defined on or throw an error.

```python
ufl_domains()
```
Return all domains this expression is defined on.

```python
static ufl_enable_profiling()
```
Turn on object counting mechanism and reset counts to zero.
class ufl.classes.Terminal
    Bases: ufl.core.expr.Expr
    
    A terminal node in the UFL expression tree.

    evaluate (x, mapping, component, index_values, derivatives=())
    Get self from mapping and return the component asked for.

    ufl_domains ()
    Return tuple of domains related to this terminal object.

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_operands = ()

class ufl.classes.FormArgument
    Bases: ufl.core.terminal.Terminal

class ufl.classes.GeometricQuantity (domain)
    Bases: ufl.core.terminal.Terminal

    is_cellwise_constant ()
    Return whether this expression is spatially constant over each cell (or over each facet for facet quantities).

    ufl_domains ()

    ufl_shape = ()

class ufl.classes.GeometricCellQuantity (domain)
    Bases: ufl.geometry.GeometricQuantity

class ufl.classes.GeometricFacetQuantity (domain)
    Bases: ufl.geometry.GeometricQuantity

class ufl.classes.SpatialCoordinate (domain)
    Bases: ufl.geometry.GeometricCellQuantity

    UFL geometry representation: The coordinate in a domain.

    In the context of expression integration, represents the domain coordinate of each quadrature point.

    In the context of expression evaluation in a point, represents the value of that point.

    evaluate (x, mapping, component, index_values)

    is_cellwise_constant ()
    Return whether this expression is spatially constant over each cell.

    name = 'x'

    ufl_shape

class ufl.classes.CellCoordinate (domain)
    Bases: ufl.geometry.GeometricCellQuantity

    UFL geometry representation: The coordinate in a reference cell.

    In the context of expression integration, represents the reference cell coordinate of each quadrature point.

    In the context of expression evaluation in a point in a cell, represents that point in the reference coordinate system of the cell.

    is_cellwise_constant ()
    Return whether this expression is spatially constant over each cell.
name = ‘X’

ufl_shape

class ufl.classes.FacetCoordinate(domain)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The coordinate in a reference cell of a facet.

In the context of expression integration over a facet, represents the reference facet coordinate of each quadrature point.

In the context of expression evaluation in a point on a facet, represents that point in the reference coordinate system of the facet.

is_cellwise_constant()
Return whether this expression is spatially constant over each cell.

name = ‘Xf’

ufl_shape

class ufl.classes.CellOrigin(domain)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The spatial coordinate corresponding to origin of a reference cell.

is_cellwise_constant()
name = ‘x0’

ufl_shape

class ufl.classes.FacetOrigin(domain)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The spatial coordinate corresponding to origin of a reference facet.

name = ‘x0f’

ufl_shape

class ufl.classes.CellFacetOrigin(domain)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The reference cell coordinate corresponding to origin of a reference facet.

name = ‘X0f’

ufl_shape

class ufl.classes.Jacobian(domain)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The Jacobian of the mapping from reference cell to spatial coordinates.

\[ J_{ij} = \frac{dx_i}{dX_j} \]

is_cellwise_constant()
Return whether this expression is spatially constant over each cell.

name = ‘J’

ufl_shape

class ufl.classes.FacetJacobian(domain)
Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The Jacobian of the mapping from reference facet to spatial coordinates.

\[ F_{ij} = \frac{dx_i}{dX_f_j} \]

The FacetJacobian is the product of the Jacobian and CellFacetJacobian:

\[ F_{J} = \frac{dx}{dX_f} = \frac{dx}{dX} \frac{dX}{dX_f} = J \cdot CFJ \]

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = ‘FJ’

ufl_shape

class ufl.classes.CellFacetJacobian(domain)

Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The Jacobian of the mapping from reference facet to reference cell coordinates.

\[ CF_{ij} = \frac{dX_i}{dX_f_j} \]

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = ‘CFJ’

ufl_shape

class ufl.classes.CellEdgeVectors(domain)

Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The vectors between reference cell vertices for each edge in cell.

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = ‘CEV’

ufl_shape

class ufl.classes.FacetEdgeVectors(domain)

Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The vectors between reference cell vertices for each edge in current facet.

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = ‘FEV’

ufl_shape

class ufl.classes.JacobianDeterminant(domain)

Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The determinant of the Jacobian.

Represents the signed determinant of a square Jacobian or the pseudo-determinant of a non-square Jacobian.

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = ‘detJ’

class ufl.classes.FacetJacobianDeterminant(domain)

Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The pseudo-determinant of the FacetJacobian.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘detFJ’}

\texttt{class ufl.classes.CellFacetJacobianDeterminant(domain)}
\texttt{Bases: ufl.geometry.GeometricFacetQuantity}
UFL geometry representation: The pseudo-determinant of the CellFacetJacobian.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘detCFJ’}

\texttt{class ufl.classes.JacobianInverse(domain)}
\texttt{Bases: ufl.geometry.GeometricCellQuantity}
UFL geometry representation: The inverse of the Jacobian.
Represents the inverse of a square Jacobian or the pseudo-inverse of a non-square Jacobian.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘K’}

\texttt{ufl\_shape}

\texttt{class ufl.classes.FacetJacobianInverse(domain)}
\texttt{Bases: ufl.geometry.GeometricFacetQuantity}
UFL geometry representation: The pseudo-inverse of the FacetJacobian.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘FK’}

\texttt{ufl\_shape}

\texttt{class ufl.classes.CellFacetJacobianInverse(domain)}
\texttt{Bases: ufl.geometry.GeometricFacetQuantity}
UFL geometry representation: The pseudo-inverse of the CellFacetJacobian.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘CFK’}

\texttt{ufl\_shape}

\texttt{class ufl.classes.FacetNormal(domain)}
\texttt{Bases: ufl.geometry.GeometricFacetQuantity}
UFL geometry representation: The outwards pointing normal vector of the current facet.

\texttt{is\_cellwise\_constant}()
Return whether this expression is spatially constant over each cell.

\texttt{name = ‘n’}

\texttt{ufl\_shape}
class ufl.classes.CellNormal (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The upwards pointing normal vector of the current manifold cell.
   name = 'cell_normal'
   ufl_shape

class ufl.classes.ReferenceNormal (domain)
   Bases: ufl.geometry.GeometricFacetQuantity
   UFL geometry representation: The outwards pointing normal vector of the current facet on the reference cell
   name = 'reference_normal'
   ufl_shape

class ufl.classes.ReferenceCellVolume (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The volume of the reference cell.
   name = 'reference_cell_volume'

class ufl.classes.ReferenceFacetVolume (domain)
   Bases: ufl.geometry.GeometricFacetQuantity
   UFL geometry representation: The volume of the reference cell of the current facet.
   name = 'reference_facet_volume'

class ufl.classes.CellVolume (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The volume of the cell.
   name = 'volume'

class ufl.classes.Circumradius (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The circumradius of the cell.
   name = 'circumradius'

class ufl.classes.FacetArea (domain)
   Bases: ufl.geometry.GeometricFacetQuantity
   UFL geometry representation: The area of the facet.
   name = 'facetarea'

class ufl.classes.MinCellEdgeLength (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The minimum edge length of the cell.
   name = 'mincelledgelength'

class ufl.classes.MaxCellEdgeLength (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The maximum edge length of the cell.
   name = 'maxcelledgelength'
class ufl.classes.MinFacetEdgeLength(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The minimum edge length of the facet.
    name = 'minfacetedgelength'

class ufl.classes.MaxFacetEdgeLength(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The maximum edge length of the facet.
    name = 'maxfacetedgelength'

class ufl.classes.CellOrientation(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The orientation (+1/-1) of the current cell.
    For non-manifold cells (tdim == gdim), this equals the sign of the Jacobian determinant, i.e. +1 if the physical
    cell is oriented the same way as the reference cell and -1 otherwise.
    For manifold cells of tdim==gdim-1 this is input data belonging to the mesh, used to distinguish between the
    sides of the manifold.
    name = 'cell_orientation'

class ufl.classes.FacetOrientation(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The orientation (+1/-1) of the current facet relative to the reference cell.
    name = 'facet_orientation'

class ufl.classes.QuadratureWeight(domain)
    Bases: ufl.geometry.GeometricQuantity
    UFL geometry representation: The current quadrature weight.
    Only used inside a quadrature context.
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
    name = 'weight'

class ufl.classes.Operator(operands=None)
    Bases: ufl.core.expr.Expr
    ufl_operands

class ufl.classes.MultiIndex(indices)
    Bases: ufl.core.terminal.Terminal
    Represents a sequence of indices, either fixed or free.
    evaluate(x, mapping, component, index_values)
    indices()
    is_cellwise_constant()
    ufl_domains()
        Return tuple of domains related to this terminal object.
    ufl_free_indices
    ufl_index_dimensions
class ConstantValue
    Bases: ufl.core_terminal.Terminal
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
    ufl_domains()
        Return tuple of domains related to this terminal object.

class Zero(shape=(), free_indices=(), index_dimensions=None)
    Bases: ufl.constantvalue.ConstantValue
    UFL literal type: Representation of a zero valued expression.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ScalarValue(value)
    Bases: ufl.constantvalue.ConstantValue
    A constant scalar value.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()
    value()

class FloatValue(value)
    Bases: ufl.constantvalue.ScalarValue
    UFL literal type: Representation of a constant scalar floating point value.

class IntValue(value)
    Bases: ufl.constantvalue.ScalarValue
    UFL literal type: Representation of a constant scalar integer value.

class Identity(dim)
    Bases: ufl.constantvalue.ConstantValue
    UFL literal type: Representation of an identity matrix.
    evaluate(x, mapping, component, index_values)
    ufl_shape

class PermutationSymbol(dim)
    Bases: ufl.constantvalue.ConstantValue
    UFL literal type: Representation of a permutation symbol.
    This is also known as the Levi-Civita symbol, antisymmetric symbol, or alternating symbol.
    evaluate(x, mapping, component, index_values)
    ufl_shape
class `ufl.classes.Indexed` *(expression, multiindex)*
   Bases: `ufl.core.operator.Operator`

   `evaluate (x, mapping, component, index_values, derivatives=())`
   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape = ()`

class `ufl.classes.ListTensor` (*expressions*)
   Bases: `ufl.core.operator.Operator`

   UFL operator type: Wraps a list of expressions into a tensor valued expression of one higher rank.
   `evaluate (x, mapping, component, index_values, derivatives=())`
   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`

class `ufl.classes.ComponentTensor` *(expression, indices)*
   Bases: `ufl.core.operator.Operator`

   UFL operator type: Maps the free indices of a scalar valued expression to tensor axes.
   `evaluate (x, mapping, component, index_values)`
   `indices ()`
   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`

class `ufl.classes.Argument` *(function_space, number, part=None)*
   Bases: `ufl.core.terminal.FormArgument`

   UFL value: Representation of an argument to a form.
   `element ()`
   `is_cellwise_constant ()`
      Return whether this expression is spatially constant over each cell.
   `number ()`
   `part ()`
   `ufl_domain ()`
   `ufl_domains ()`
      Return tuple of domains related to this terminal object.
   `ufl_element ()`
   `ufl_function_space ()`
      Get the function space of this Argument.
   `ufl_shape`

class `ufl.classes.Coefficient` *(function_space, count=None)*
   Bases: `ufl.core.terminal.FormArgument`

   UFL form argument type: Representation of a form coefficient.
count()

element()

is_cellwise_constant()
    Return whether this expression is spatially constant over each cell.

ufl_domain()
    Shortcut to get the domain of the function space of this coefficient.

ufl_domains()
    Return tuple of domains related to this terminal object.

ufl_element()
    Shortcut to get the finite element of the function space of this coefficient.

ufl_function_space()
    Get the function space of this coefficient.

class ufl.classes.Label(count=None)
    Bases: ufl.core.terminal.Terminal

    count()
    is_cellwise_constant()
    ufl_domains()
        Return tuple of domains related to this terminal object.
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Variable(expression, label=None)
    Bases: ufl.core.operator.Operator

    A Variable is a representative for another expression.
    It will be used by the end-user mainly for defining a quantity to differentiate w.r.t. using `diff`. Example:

    ```
    e = <...>
    e = variable(e)
    f = exp(e**2)
    df = diff(f, e)
    ```

evaluate(x, mapping, component, index_values)

    expression()
    label()
    ufl_domains()
        ufl_free_indices =()
        ufl_index_dimensions =()
        ufl_shape

class ufl.classes.Sum(a, b)
    Bases: ufl.core.operator.Operator

    evaluate(x, mapping, component, index_values)
class ufl.classes.Product(a, b)
    Bases: ufl.core.operator.Operator
    The product of two or more UFL objects.
    
    evaluate(x, mapping, component, index_values)
    
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.classes.Discrition(a, b)
    Bases: ufl.core.operator.Operator
    
    evaluate(x, mapping, component, index_values)
    
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.classes.Power(a, b)
    Bases: ufl.core.operator.Operator
    
    evaluate(x, mapping, component, index_values)
    
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.classes.Abs(a)
    Bases: ufl.core.operator.Operator
    
    evaluate(x, mapping, component, index_values)
    
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.CompoundTensorOperator(operands)
    Bases: ufl.core.operator.Operator

class ufl.classes.Transposed(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Outer(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    
    ufl_free_indices
class ufl.classes.Inner(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.classes.Dot(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Cross(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = (3,)

class ufl.classes.Trace(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.classes.Determinant(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.classes.Inverse(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape

class ufl.classes.Cofactor(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape

class ufl.classes.Deviatoric(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
class ufl.classes.Skew(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Sym(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.IndexSum(summand, index)
    Bases: ufl.core.operator.Operator
dimension()
    evaluate(x, mapping, component, index_values)
    index()
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Restricted(f)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
    side()
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.PositiveRestricted(f)
    Bases: ufl.restriction.Restricted

class ufl.classes.NegativeRestricted(f)
    Bases: ufl.restriction.Restricted

class ufl.classes.CellAvg(f)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
    Performs an approximate symbolic evaluation, since we dont have a cell.
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape
class `ufl.classes.FacetAvg(f)`
   Bases: `ufl.core.operator.Operator`

   `evaluate(x, mapping, component, index_values)`
   Performs an approximate symbolic evaluation, since we don't have a cell.

   `ufl_free_indices = ()`
   `ufl_index_dimensions = ()`
   `ufl_shape`

class `ufl.classes.ExprList(*operands)`
   Bases: `ufl.core.operator.Operator`

   List of Expr objects. For internal use, never to be created by end users.

   `free_indices()`
   `index_dimensions()`
   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`

class `ufl.classes.ExprMapping(*operands)`
   Bases: `ufl.core.operator.Operator`

   Mapping of Expr objects. For internal use, never to be created by end users.

   `free_indices()`
   `index_dimensions()`
   `ufl_domains()`
   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`

class `ufl.classes.Derivative(operands)`
   Bases: `ufl.core.operator.Operator`

   Base class for all derivative types.

class `ufl.classes.CoefficientDerivative(integrand, coefficients, arguments, coefficient_derivatives)`
   Bases: `ufl.differentiation.Derivative`

   Derivative of the integrand of a form w.r.t. the degrees of freedom in a discrete Coefficient.

   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`

class `ufl.classes.VariableDerivative(f, v)`
   Bases: `ufl.differentiation.Derivative`

   `ufl_free_indices`
   `ufl_index_dimensions`
   `ufl_shape`
class ufl.classes.CompositeDerivative(operands)
   Bases: ufl.differentiation.Derivative

   Base class for all compound derivative types.

class ufl.classes.Grad(f)
   Bases: ufl.differentiation.CompositeDerivative

evaluate(x, mapping, component, index_values, derivatives=())
   Get child from mapping and return the component asked for.

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.ReferenceGrad(f)
   Bases: ufl.differentiation.CompositeDerivative

   evaluate(x, mapping, component, index_values, derivatives=())
   Get child from mapping and return the component asked for.

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.Div(f)
   Bases: ufl.differentiation.CompositeDerivative

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.ReferenceDiv(f)
   Bases: ufl.differentiation.CompositeDerivative

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.NablaGrad(f)
   Bases: ufl.differentiation.CompositeDerivative

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.NablaDiv(f)
   Bases: ufl.differentiation.CompositeDerivative

   ufl_free_indices
   ufl_index_dimensions
   ufl_shape

class ufl.classes.Curl(f)
   Bases: ufl.differentiation.CompositeDerivative
class ufl.classes.ReferenceCurl(f)  
Bases: ufl.differentiation.CompoundDerivative

class ufl.classes.Condition(operands)  
Bases: ufl.core.operator.Operator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.classes.BinaryCondition(name, left, right)  
Bases: ufl.conditional.Condition

class ufl.classes.EQ(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.NE(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.LE(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.GE(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.LT(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.GT(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.AndCondition(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.classes.OrCondition(left, right)  
Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)
class ufl.classes.NotCondition(condition)
    Bases: ufl.conditional.Condition
    evaluate(x, mapping, component, index_values)

class ufl.classes.Conditional(condition, true_value, false_value)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.MinValue(left, right)
    Bases: ufl.core.operator.Operator
    UFL operator: Take the minimum of two values.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.classes.MaxValue(left, right)
    Bases: ufl.core.operator.Operator
    UFL operator: Take the maximum of two values.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.classes.MathFunction(name, argument)
    Bases: ufl.core.operator.Operator
    Base class for all unary scalar math functions.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.classes.Sqrt(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Exp(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Ln(argument)
    Bases: ufl.mathfunctions.MathFunction
    evaluate(x, mapping, component, index_values)

class ufl.classes.Cos(argument)
    Bases: ufl.mathfunctions.MathFunction
class ufl.classes.Sin(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Tan(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Cosh(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Sinh(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Tanh(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Acos(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Asin(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Atan(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Atan2(arg1, arg2)
    Bases: ufl.core.operator.Operator

    evaluate(x, mapping, component, index_values)

    ufl_free_indices = ()

    ufl_index_dimensions = ()

    ufl_shape = ()

class ufl.classes.Erf(argument)
    Bases: ufl.mathfunctions.MathFunction

    evaluate(x, mapping, component, index_values)

class ufl.classes.BesselFunction(name, classname, nu, argument)
    Bases: ufl.core.operator.Operator

    Base class for all bessel functions

    evaluate(x, mapping, component, index_values)

    ufl_free_indices = ()

    ufl_index_dimensions = ()

    ufl_shape = ()

class ufl.classes.BesselJ(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.classes.BesselY(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.classes.BesselI(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.classes.BesselK(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction
**class** ufl.classes.ReferenceValue(f)

**Bases:** ufl.core.operator.Operator

Representation of the reference cell value of a form argument.

**evaluate**(x, mapping, component, index_values, derivatives=())

Get child from mapping and return the component asked for.

**ufl_free_indices** = ()

**ufl_index_dimensions** = ()

**ufl_shape**

**class** ufl.classes.AbstractCell(topological_dimension, geometric_dimension)

**Bases:** object

Representation of an abstract finite element cell with only the dimensions known.

**geometric_dimension**()

Return the dimension of the space this cell is embedded in.

**has_simplex_facets**()

Return True if all the facets of this cell are simplex cells.

**is_simplex**()

Return True if this is a simplex cell.

**topological_dimension**()

Return the dimension of the topology of this cell.

**class** ufl.classes.Cell(cellname, geometric_dimension=None)

**Bases:** ufl.cell.AbstractCell

Representation of a named finite element cell with known structure.

**cellname**()

Return the cellname of the cell.

**has_simplex_facets**()

**is_simplex**()

**num_edges**()

The number of cell edges.

**num_facet_edges**()

The number of facet edges.

**num_facets**()

The number of cell facets.

**num_vertices**()

The number of cell vertices.

**class** ufl.classes.TensorProductCell(cells)

**Bases:** ufl.cell.AbstractCell

**has_simplex_facets**()

Return True if all the facets of this cell are simplex cells.

**is_simplex**()

Return True if this is a simplex cell.

**num_edges**()

The number of cell edges.
num_facets()
    The number of cell facets.

num_vertices()
    The number of cell vertices.

sub_cells()
    Return list of cell factors.

class ufl.classes.OuterProductCell(A, B, gdim=None)
    Bases: ufl.cell.AbstractCell
    Representation of a cell formed as the Cartesian product of two existing cells

    facet_horiz
    facet_vert

    has_simplex_facets()
    Return True if all the facets of this cell are simplex cells.

    is_simplex()
    Return True if this is a simplex cell.

num_edges()
    The number of cell edges.

num_facets()
    The number of cell facets.

num_vertices()
    The number of cell vertices.

class ufl.classes.FiniteElementBase(family, cell, degree, quad_scheme, value_shape, reference_value_shape)
    Bases: object
    Base class for all finite elements

    cell()
    Return cell of finite element

    degree(component=None)
    Return polynomial degree of finite element

    extract_component(i)
    Recursively extract component index relative to a (simple) element and that element for given value component index

    extract_reference_component(i)
    Recursively extract reference component index relative to a (simple) element and that element for given reference value component index

    extract_subelement_component(i)
    Extract direct subelement index and subelement relative component index for a given component index

    extract_subelement_reference_component(i)
    Extract direct subelement index and subelement relative reference component index for a given reference component index

    family()
    Return finite element family

    is_cellwise_constant(component=None)
    Return whether the basis functions of this element is spatially constant over each cell.
mapping()

num_sub_elements()
    Return number of sub elements

quadrature_scheme()
    Return quadrature scheme of finite element

reference_value_shape()
    Return the shape of the value space on the reference cell.

reference_value_size()
    Return the integer product of the reference value shape.

sub_elements()
    Return list of sub elements

symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by
    component c1.

value_shape()
    Return the shape of the value space on the global domain.

value_size()
    Return the integer product of the value shape.

class ufl.classes.FiniteElement (family, cell=None, degree=None, form_degree=None, quad_scheme=None)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
    The basic finite element class for all simple finite elements

mapping()

shortstr()
    Format as string for pretty printing.

sobolev_space()

class ufl.classes.MixedElement (*elements, **kwargs)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
    A finite element composed of a nested hierarchy of mixed or simple elements

degree (component=None)
    Return polynomial degree of finite element

extract_component (i)
    Recursively extract component index relative to a (simple) element and that element for given value com-
    ponent index

extract_reference_component (i)
    Recursively extract reference_component index relative to a (simple) element and that element for given
    value reference_component index

extract_subelement_component (i)
    Extract direct subelement index and subelement relative component index for a given component index

extract_subelement_reference_component (i)
    Extract direct subelement index and subelement relative reference_component index for a given reference
    component index

is_cellwise_constant (component=None)
    Return whether the basis functions of this element is spatially constant over each cell.
mapping()

num_sub_elements()
    Return number of sub elements.

reconstruct_from_elements(*elements)
    Reconstruct a mixed element from new subelements.

shortstr()
    Format as string for pretty printing.

sub_elements()
    Return list of sub elements.

symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1. A component is a tuple of one or more ints.

class ufl.classes.VectorElement (family, cell, degree, dim=None, form_degree=None, quad_scheme=None)
    Bases: ufl.finiteelement.mixedelement.MixedElement

    A special case of a mixed finite element where all elements are equal

shortstr()
    Format as string for pretty printing.

class ufl.classes.TensorElement (family, cell, degree, shape=None, symmetry=None, quad_scheme=None)
    Bases: ufl.finiteelement.mixedelement.MixedElement

    A special case of a mixed finite element where all elements are equal

extract_subelement_component(i)
    Extract direct subelement index and subelement relative component index for a given component index

flattened_sub_element_mapping()

mapping()

shortstr()
    Format as string for pretty printing.

symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

class ufl.classes.EnrichedElement (*elements)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    The vector sum of two finite element spaces:
    EnrichedElement(V, Q) = {v + q | v in V, q in Q}.

    is_cellwise_constant()
        Return whether the basis functions of this element is spatially constant over each cell.

mapping()

shortstr()
    Format as string for pretty printing.

class ufl.classes.RestrictedElement (element, restriction_domain)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    Represents the restriction of a finite element to a type of cell entity.
element()

is_cellwise_constant()
    Return whether the basis functions of this element is spatially constant over each cell.

mapping()

num_restricted_sub_elements()
    Return number of restricted sub elements.

num_sub_elements()
    Return number of sub elements

restricted_sub_elements()
    Return list of restricted sub elements.

restriction_domain()
    Return the domain onto which the element is restricted.

shortstr()
    Format as string for pretty printing.

sub_element()
    Return the element which is restricted.

sub_elements()
    Return list of sub elements

symmetry()
    Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

class ufl.classes.TensorProductElement(elements)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The tensor product of d element spaces:

\[ V = V_0 \otimes V_1 \otimes ... \otimes V_d \]

Given bases \{\phi_i\} for \(V_i\) for i = 1, ..., d, \{ \phi_0 * \phi_1 * ... * \phi_d \} forms a basis for V.

mapping()

num_sub_elements()
    Return number of subelements.

shortstr()
    Short pretty-print.

sub_elements()
    Return subelements (factors).

class ufl.classes.OuterProductElement(A, B, cell=None)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The outer (tensor) product of 2 element spaces:

\[ V = A \otimes B \]

Given bases \(\phi_A, \phi_B\) for A, B, \(\phi_A * \phi_B\) forms a basis for V.

mapping()

shortstr()
    Short pretty-print.
class ufl.classes.OuterProductVectorElement (A, B, cell=None, dim=None)
   Bases: ufl.finiteelement.mixedelement.MixedElement
   A special case of a mixed finite element where all elements are equal OuterProductElements
   mapping()
   shortstr()
      Format as string for pretty printing.

class ufl.classes.HDivElement (element)
   Bases: ufl.finiteelement.outerproductelement.OuterProductElement
   A div-conforming version of an outer product element, assuming this makes mathematical sense.
   shortstr()

class ufl.classes.HCurlElement (element)
   Bases: ufl.finiteelement.outerproductelement.OuterProductElement
   A curl-conforming version of an outer product element, assuming this makes mathematical sense.
   shortstr()

class ufl.classes.BrokenElement (element)
   Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
   The discontinuous version of an existing Finite Element space
   mapping()
   shortstr()

class ufl.classes.TraceElement (element)
   Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
   A finite element space: the trace of a given hdiv element. This is effectively a scalar-valued restriction which is
   non-zero only on cell facets.
   mapping()
   shortstr()

class ufl.classes.FacetElement (element)
   Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
   A version of an existing Finite Element space in which all dofs associated with the interior have been discarded
   mapping()
   shortstr()

class ufl.classes.InteriorElement (element)
   Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
   A version of an existing Finite Element space in which only the dofs associated with the interior have been kept
   mapping()
   shortstr()

class ufl.classes.AbstractDomain (topological_dimension, geometric_dimension)
   Bases: object
   Symbolic representation of a geometric domain with only a geometric and topological dimension.
geometric_dimension()
    Return the dimension of the space this domain is embedded in.

topological_dimension()
    Return the dimension of the topology of this domain.

class ufl.classes.Mesh(coordinate_element, ufl_id=None, cargo=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    cell()
    coordinates()
    is_piecewise_linear_simplex_domain()
    ufl_cargo()
        Return carried object that will not be used by UFL.
    ufl_cell()
    ufl_coordinate_element()
    ufl_coordinates()
    ufl_id()
        Return the ufl_id of this object.

class ufl.classes.MeshView(mesh, topological_dimension, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_id()
        Return the ufl_id of this object.
    ufl_mesh()

class ufl.classes.TensorProductMesh(meshes, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_coordinate_element()
    ufl_id()
        Return the ufl_id of this object.

class ufl.classes.AbstractFunctionSpace
    Bases: object
    ufl_sub_spaces()

class ufl.classes.FunctionSpace(domain, element)
    Bases: ufl.functionspace.AbstractFunctionSpace
    ufl_domain()
    ufl_element()
unified_form_language (UFL) Documentation, Release 1.7.0dev

ufl_sub_spaces()

class ufl.classes.MixedFunctionSpace(*function_spaces)
   Bases: ufl.functionspace.AbstractFunctionSpace
   ufl_sub_spaces()

class ufl.classes.TensorProductFunctionSpace(*function_spaces)
   Bases: ufl.functionspace.AbstractFunctionSpace
   ufl_sub_spaces()

class ufl.classes.IndexBase
   Bases: object

class ufl.classes.FixedIndex(value)
   Bases: ufl.core.multiindex.IndexBase
   UFL value: An index with a specific value assigned.

class ufl.classes.Index(count=None)
   Bases: ufl.core.multiindex.IndexBase
   UFL value: An index with no value assigned.
   Used to represent free indices in Einstein indexing notation.
   count()

class ufl.classes.TestFunction(function_space, part=None)
   UFL value: Create a test function argument to a form.

class ufl.classes.TrialFunction(function_space, part=None)
   UFL value: Create a trial function argument to a form.

class ufl.classes.TestFunctions(function_space)
   UFL value: Create a TestFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

class ufl.classes.TrialFunctions(function_space)
   UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

class ufl.classes.Measure(integral_type, domain=None, subdomain_id='everywhere', metadata=None, subdomain_data=None)
   Bases: object
   domain()
   integral_type()
      Return the domain type.
      Valid domain types are “cell”, “exterior_facet”, “interior_facet”, etc.
   metadata()
      Return the integral metadata. This data is not interpreted by UFL. It is passed to the form compiler which can ignore it or use it to compile each integral of a form in a different way.
   reconstruct(integral_type=None, subdomain_id=None, domain=None, metadata=None, subdomain_data=None)
      Construct a new Measure object with some properties replaced with new values.
      Example: <dm = Measure instance> b = dm.reconstruct(subdomain_id=2) c = dm.reconstruct(metadata={ "quadrature_degree": 3 })
      Used by the call operator, so this is equivalent: b = dm(2) c = dm(0, { "quadrature_degree": 3 })

2.2. ufl package
subdomain_data()

Return the integral subdomain_data. This data is not interpreted by UFL. Its intension is to give a context in which the domain id is interpreted.

subdomain_id()

Return the domain id of this measure (integer).

ufl_domain()

Return the domain associated with this measure.

This may be None or a Domain object.

class ufl.classes.MeasureSum(*measures)

Bases: object

Represents a sum of measures.

This is a notational intermediate object to translate the notation

\[ f*(ds(1)+ds(3)) \]

into

\[ f*ds(1) + f*ds(3) \]

class ufl.classes.MeasureProduct(*measures)

Bases: object

Represents a product of measures.

This is a notational intermediate object to handle the notation

\[ f*(dm1*dm2) \]

This is work in progress and not functional. It needs support in other parts of ufl and the rest of the code generation chain.

sub_measures()

Return submeasures.

class ufl.classes.Integral(integrand, integral_type, domain, subdomain_id, metadata, subdomain_data)

Bases: object

An integral over a single domain.

domain()

integral_type()

Return the domain type of this integral.

integrand()

Return the integrand expression, which is an Expr instance.

metadata()

Return the compiler metadata this integral has been annotated with.

reconstruct(integrand=None, integral_type=None, domain=None, subdomain_id=None, metadata=None, subdomain_data=None)

Construct a new Integral object with some properties replaced with new values.

Example:

```python
a = Integral instance
b = a.reconstruct(expand_compounds(a.integrand()))
c = a.reconstruct(metadata={'quadrature_degree':2})
```

subdomain_data()

Return the domain data of this integral.
subdomain_id()
    Return the subdomain id of this integral.

ufl_domain()
    Return the integration domain of this integral.

class ufl.classes.Form(integrals)
    Bases: object
    Description of a weak form consisting of a sum of integrals over subdomains.

    arguments()
        Return all Argument objects found in form.

cell()

coefficient_numbering()
    Return a contiguous numbering of coefficients in a mapping { coefficient: number }.

coefficients()
    Return all Coefficient objects found in form.

domain()

domain_numbering()
    Return a contiguous numbering of domains in a mapping { domain: number }.

domains()

empty()
    Returns whether the form has no integrals.

equals(other)
    Evaluate ‘bool(lhs_form == rhs_form)”.

geometric_dimension()
    Return the geometric dimension shared by all domains and functions in this form.

integrals()
    Return a sequence of all integrals in form.

integrals_by_type(integral_type)
    Return a sequence of all integrals with a particular domain type.

max_subdomain_ids()
    Returns a mapping on the form { domain: { integral_type: max_subdomain_id } }.

signature()
    Signature for use with jit cache (independent of incidental numbering of indices etc.)

subdomain_data()
    Returns a mapping on the form { domain: { integral_type: subdomain_data } }.

ufl_cell()
    Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()
    Return the single geometric integration domain occuring in the form.

    Fails if multiple domains are found.

    NB! This does not include domains of coefficients defined on other meshes, look at form data for that additional information.
ufl_domains()
Return the geometric integration domains occurring in the form.
NB! This does not include domains of coefficients defined on other meshes.
The return type is a tuple even if only a single domain exists.
x_repr_latex()
x_repr_png()

class ufl.classes.Equation(lhs, rhs)
This class is used to represent equations expressed by the “==” operator. Examples include a == L and F == 0
where a, L and F are Form objects.

2.2.9 ufl.coefficient module
This module defines the Coefficient class and a number of related classes, including Constant.
class ufl.coefficient.Coefficient(function_space, count=None)
Bases: ufl.core.terminal.FormArgument
UFL form argument type: Representation of a form coefficient.
count()
element()
is_cellwise_constant()
Return whether this expression is spatially constant over each cell.
ufl_domain()
Shortcut to get the domain of the function space of this coefficient.
ufl_domains()
Return tuple of domains related to this terminal object.
ufl_element()
Shortcut to get the finite element of the function space of this coefficient.
ufl_function_space()
Get the function space of this coefficient.
ufl_shape

ufl.coefficient.Coefficients(function_space)
UFL value: Create a Coefficient in a mixed space, and return a tuple with the function components corresponding
to the subelements.
ufl.coefficient.Constant(domain, count=None)
UFL value: Represents a globally constant scalar valued coefficient.
ufl.coefficient.TensorConstant(domain, shape=None, symmetry=None, count=None)
UFL value: Represents a globally constant tensor valued coefficient.
ufl.coefficient.VectorConstant(domain, dim=None, count=None)
UFL value: Represents a globally constant vector valued coefficient.

2.2.10 ufl.compound_expressions module
Functions implementing compound expressions as equivalent representations using basic operators.
Unified Form Language (UFL) Documentation, Release 1.7.0dev

ufl.compound_expressions.adj_expr(A)
ufl.compound_expressions.adj_expr_2x2(A)
ufl.compound_expressions.adj_expr_3x3(A)
ufl.compound_expressions.adj_expr_4x4(A)
ufl.compound_expressions.codeterminant_expr_nxn(A, rows, cols)
ufl.compound_expressions.cofactor_expr(A)
ufl.compound_expressions.cofactor_expr_2x2(A)
ufl.compound_expressions.cofactor_expr_3x3(A)
ufl.compound_expressions.cofactor_expr_4x4(A)
ufl.compound_expressions.cross_expr(a, b)
ufl.compound_expressions.determinant_expr(A)
   Compute the (pseudo-)determinant of A.
ufl.compound_expressions.determinant_expr_2x2(B)
ufl.compound_expressions.determinant_expr_3x3(A)
ufl.compound_expressions.deviatoric_expr(A)
ufl.compound_expressions.deviatoric_expr_2x2(A)
ufl.compound_expressions.deviatoric_expr_3x3(A)
ufl.compound_expressions.generic_pseudo_determinant_expr(A)
   Compute the pseudo-determinant of A: sqrt(det(A.T*A)).
ufl.compound_expressions.generic_pseudo_inverse_expr(A)
   Compute the Penrose-Moore pseudo-inverse of A: (A.T*A)^-1 * A.T.
ufl.compound_expressions.inverse_expr(A)
   Compute the inverse of A.
ufl.compound_expressions.old_determinant_expr_3x3(A)
ufl.compound_expressions.pseudo_determinant_expr(A)
   Compute the pseudo-determinant of A.
ufl.compound_expressions.pseudo_inverse_expr(A)
   Compute the Penrose-Moore pseudo-inverse of A: (A.T*A)^-1 * A.T.

2.2.11 ufl.conditional module

This module defines classes for conditional expressions.

```python
class ufl.conditional.AndCondition(left, right)
   Bases: ufl.conditional.BinaryCondition
   evaluate(x, mapping, component, index_values)

class ufl.conditional.BinaryCondition(name, left, right)
   Bases: ufl.conditional.Condition

class ufl.conditional.Condition(operands)
   Bases: ufl.core.operator.Operator
   ufl_free_indices = ()
```

2.2. ufl package | 131
class ufl.conditional.Conditional(condition, true_value, false_value)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)

class ufl.conditional.EQ(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.conditional.GE(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.conditional.GT(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.conditional.LE(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.conditional.LT(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)

class ufl.conditional.MaxValue(left, right)
    Bases: ufl.core.operator.Operator
    UFL operator: Take the maximum of two values.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.conditional.MinValue(left, right)
    Bases: ufl.core.operator.Operator
    UFL operator: Take the minimum of two values.
    evaluate(x, mapping, component, index_values)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.conditional.NE(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)

class ufl.conditional.NotCondition(condition)
    Bases: ufl.conditional.Condition
    evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)

class ufl.conditional.OrCondition(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)

2.2.12 ufl.constantvalue module

This module defines classes representing constant values.

class ufl.constantvalue.ConstantValue
    Bases: ufl.core.terminal.Terminal
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
    ufl_domains()
        Return tuple of domains related to this terminal object.

class ufl.constantvalue.FloatValue(value)
    Bases: ufl.constantvalue.ScalarValue
    UFL literal type: Representation of a constant scalar floating point value.

class ufl.constantvalue.Identity(dim)
    Bases: ufl.constantvalue.ConstantValue
    UFL literal type: Representation of an identity matrix.
    evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
    ufl_shape

class ufl.constantvalue.IntValue(value)
    Bases: ufl.constantvalue.ScalarValue
    UFL literal type: Representation of a constant scalar integer value.

class ufl.constantvalue.PermutationSymbol(dim)
    Bases: ufl.constantvalue.ConstantValue
    UFL literal type: Representation of a permutation symbol.
    This is also known as the Levi-Civita symbol, antisymmetric symbol, or alternating symbol.
    evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
    ufl_shape

class ufl.constantvalue.ScalarValue(value)
    Bases: ufl.constantvalue.ConstantValue
    A constant scalar value.
    evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
    ufl_free_indices = ()
    ufl_index_dimensions = ()
class ufl.constantvalue.Zero(shape=(), free_indices=(), index_dimensions=None)
Bases: ufl.constantvalue.ConstantValue
UFL literal type: Representation of a zero valued expression.
evaluate(x, mapping, component, index_values)

ufl_free_indices
ufl_index_dimensions
ufl_shape

ufl.constantvalue.as_ufl(expression)
Converts expression to an Expr if possible.
ufl.constantvalue.format_float(x)
Format float value based on global UFL precision.
ufl.constantvalue.zero(*shape)
UFL literal constant: Return a zero tensor with the given shape.

2.2.13 ufl.differentiation module

Differential operators.

class ufl.differentiation.CoefficientDerivative(integrand, coefficients, arguments, coefficient_derivatives)
Bases: ufl.differentiation.Derivative
Derivative of the integrand of a form w.r.t. the degrees of freedom in a discrete Coefficient.

ufl_free_indices
ufl_index_dimensions
ufl_shape

class ufl.differentiation.Curl(f)
Bases: ufl.differentiation.CompoundDerivative
Base class for all compound derivative types.

ufl_free_indices
ufl_index_dimensions
ufl_shape

class ufl.differentiation.Derivative(operands)
Bases: ufl.core.operator.Operator
Base class for all derivative types.

ufl_free_indices
class ufl.differentiation.Grad(f)
    Bases: ufl.differentiation.CompoundDerivative
    evaluate (x, mapping, component, index_values, derivatives=())
    Get child from mapping and return the component asked for.

class ufl.differentiation.NablaDiv(f)
    Bases: ufl.differentiation.CompoundDerivative

class ufl.differentiation.NablaGrad(f)
    Bases: ufl.differentiation.CompoundDerivative

class ufl.differentiation.ReferenceCurl(f)
    Bases: ufl.differentiation.CompoundDerivative

class ufl.differentiation.ReferenceDiv(f)
    Bases: ufl.differentiation.CompoundDerivative

class ufl.differentiation.VariableDerivative(f, v)
    Bases: ufl.differentiation.Derivative
2.2.14 ufl.domain module

Types for representing a geometric domain.

class ufl.domain.AbstractDomain(topological_dimension, geometric_dimension)
    Bases: object
    Symbolic representation of a geometric domain with only a geometric and topological dimension.
    geometric_dimension()
        Return the dimension of the space this domain is embedded in.
    topological_dimension()
        Return the dimension of the topology of this domain.

class ufl.domain.Mesh(coordinate_element, ufl_id=None, cargo=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    cell()
    coordinates()
    is_piecewise_linear_simplex_domain()
    ufl_cargo()
        Return carried object that will not be used by UFL.
    ufl_cell()
    ufl_coordinate_element()
    ufl_coordinates()
    ufl_id()
        Return the ufl_id of this object.

class ufl.domain.MeshView(mesh, topological_dimension, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_id()
        Return the ufl_id of this object.
    ufl_mesh()

class ufl.domain.TensorProductMesh(meshes, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_coordinate_element()
ufl_id()
   Return the ufl_id of this object.

ufl.domain.affine_mesh(cell, ufl_id=None)
   Create a Mesh over a given cell type with an affine geometric parameterization.

ufl.domain.as_domain(domain)
   Convert any valid object to an AbstractDomain type.

ufl.domain.default_domain(cell)
   Create a singular default Mesh from a cell, always returning the same Mesh object for the same cell.

ufl.domain.extract_domains(expr)
   Return all domains expression is defined on.

ufl.domain.extract_unique_domain(expr)
   Return the single unique domain expression is defined on or throw an error.

ufl.domain.find_geometric_dimension(expr)
   Find the geometric dimension of an expression.

ufl.domain.join_domains(domains)
   Take a list of domains and return a tuple with only unique domain objects.
   Checks that domains with the same id are compatible.

ufl.domain.sort_domains(domains)
   Sort domains in a canonical ordering.

2.2.15 ufl.equation module

The Equation class, used to express equations like a == L.

class ufl.equation.Equation(lhs, rhs)
   This class is used to represent equations expressed by the “==” operator. Examples include a == L and F == 0
   where a, L and F are Form objects.

2.2.16 ufl.exprcontainers module

This module defines special types for representing mapping of expressions to expressions.

class ufl.exprcontainers.ExprList(*operands)
   Bases: ufl.core.operator.Operator
       List of Expr objects. For internal use, never to be created by end users.

       free_indices()

       index_dimensions()

       ufl_free_indices

       ufl_index_dimensions

       ufl_shape

class ufl.exprcontainers.ExprMapping(*operands)
   Bases: ufl.core.operator.Operator
       Mapping of Expr objects. For internal use, never to be created by end users.

       free_indices()
2.2.17 ufl.exprequals module

ufl.exprequals.expr_equals (self, other)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions
are mathematically equal or equivalent! Used by sets and dicts.

ufl.exprequals.measure_collisions (equals_func)

ufl.exprequals.nonrecursive_expr_equals (self, other)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions
are mathematically equal or equivalent! Used by sets and dicts.

ufl.exprequals.print_collisions()

ufl.exprequals.recursive_expr_equals (self, other)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions
are mathematically equal or equivalent! Used by sets and dicts.

2.2.18 ufl.exproperators module

This module attaches special functions to Expr. This way we avoid circular dependencies between e.g. Sum and its
superclass Expr.

ufl.exproperators.analyse_key (ii, rank)
Takes something the user might input as an index tuple inside [], which could include complete slices (:) and
ellipsis (...), and returns tuples of actual UFL index objects.

The return value is a tuple (indices, axis_indices), each being a tuple of IndexBase instances.

The return value ‘indices’ corresponds to all input objects of these types: - Index - FixedIndex - int => Wrapped
in FixedIndex

The return value ‘axis_indices’ corresponds to all input objects of these types: - Complete slice (:) => Replaced
by a single new index - Ellipsis (...) => Replaced by multiple new indices

2.2.19 ufl.form module

The Form class.

class ufl.form.Form (integrals)
  Bases: object

  Description of a weak form consisting of a sum of integrals over subdomains.

  arguments ()
    Return all Argument objects found in form.

  cell ()
coefficient_numbering()  
Return a contiguous numbering of coefficients in a mapping { coefficient: number }.

coefficients()  
Return all Coefficient objects found in form.

domain()  
domain_numbering()  
Return a contiguous numbering of domains in a mapping { domain: number }.

domains()  
empty()  
Returns whether the form has no integrals.

equals(other)  
Evaluate ‘bool(lhs_form == rhs_form)’.

geometric_dimension()  
Return the geometric dimension shared by all domains and functions in this form.

integrals()  
Return a sequence of all integrals in form.

integrals_by_type(integral_type)  
Return a sequence of all integrals with a particular domain type.

max_subdomain_ids()  
Returns a mapping on the form { domain: { integral_type: max_subdomain_id } }.

signature()  
Signature for use with jit cache (independent of incidental numbering of indices etc.)

subdomain_data()  
Returns a mapping on the form { domain: { integral_type: subdomain_data } }.

ufl_cell()  
Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()  
Return the single geometric integration domain occurring in the form.

Fails if multiple domains are found.

NB! This does not include domains of coefficients defined on other meshes, look at form data for that additional information.

ufl_domains()  
Return the geometric integration domains occurring in the form.

NB! This does not include domains of coefficients defined on other meshes.

The return type is a tuple even if only a single domain exists.

x_repr_latex_()  
x_repr_png_()  
ufl.form.as_form(form)  
Convert to form if not a form, otherwise return form.

ufl.form.replace_integral_domains(form, common_domain)  
Given a form and a domain, assign a common integration domain to all integrals.
Does not modify the input form (Form should always be immutable). This is to support ill formed forms with no domain specified, some times occurring in pydolfin, e.g. assemble(1*dx, mesh=mesh).

## 2.2.20 ufl.formoperators module

Various high level ways to transform a complete Form into a new Form.

### ufl.formoperators.action(form, coefficient=None)

UFL form operator: Given a bilinear form, return a linear form with an additional coefficient, representing the action of the form on the coefficient. This can be used for matrix-free methods.

### ufl.formoperators.adjoint(form, reordered_arguments=None)

UFL form operator: Given a combined bilinear form, compute the adjoint form by changing the ordering (count) of the test and trial functions.

By default, new Argument objects will be created with opposite ordering. However, if the adjoint form is to be added to other forms later, their arguments must match. In that case, the user must provide a tuple reordered_arguments=(u2,v2).

### ufl.formoperators.derivative(form, coefficient, argument=None, coefficient_derivatives=None)

UFL form operator: Compute the Gateaux derivative of form w.r.t. coefficient in direction of argument.

If the argument is omitted, a new Argument is created in the same space as the coefficient, with argument number one higher than the highest one in the form.

The resulting form has one additional Argument in the same finite element space as the coefficient.

A tuple of Coefficients may be provided in place of a single Coefficient, in which case the new Argument argument is based on a MixedElement created from this tuple.

An indexed Coefficient from a mixed space may be provided, in which case the argument should be in the corresponding subspace of the coefficient space.

If provided, coefficient_derivatives should be a mapping from Coefficient instances to their derivatives w.r.t. ‘coefficient’.

### ufl.formoperators.energy_norm(form, coefficient=None)

UFL form operator: Given a bilinear form a and a coefficient f, return the functional a(f,f).

### ufl.formoperators.functional(form)

UFL form operator: Extract the functional part of form.

### ufl.formoperators.lhs(form)

UFL form operator: Given a combined bilinear and linear form, extract the left hand side (bilinear form part).

Example:

\[a = u^*v^*dx + f^*v^*dx\]

\[a = \text{lhs}(a) \Rightarrow u^*v^*dx\]

### ufl.formoperatorsrhs(form)

UFL form operator: Given a combined bilinear and linear form, extract the right hand side (negated linear form part).

Example:

\[a = u^*v^*dx + f^*v^*dx\]

\[L = \text{rhs}(a) \Rightarrow -f^*v^*dx\]

### ufl.formoperators.sensitivity_rhs(a, u, L, v)

UFL form operator: Compute the right hand side for a sensitivity calculation system.

The derivation behind this computation is as follows. Assume a, L to be bilinear and linear forms corresponding to the assembled linear system.
Ax = b.

Where x is the vector of the discrete function corresponding to u. Let v be some scalar variable this equation depends on. Then we can write

\[ 0 = \frac{d}{dv}[Ax - b] = \frac{dA}{dv} x + A \frac{dx}{dv} - \frac{db}{dv}, \quad A \frac{dx}{dv} = \frac{db}{dv} - \frac{dA}{dv} x, \]

and solve this system for \(\frac{dx}{dv}\), using the same bilinear form a and matrix A from the original system. Assume the forms are written

\[ v = \text{variable}(v_{\text{expression}}) \quad L = IL(v) * dx \quad a = Ia(v) * dx \]

where IL and Ia are integrand expressions. Define a Coefficient u representing the solution to the equations. Then we can compute \(\frac{db}{dv}\) and \(\frac{dA}{dv}\) from the forms

\[ da = \text{diff}(a, v) \quad dL = \text{diff}(L, v) \]

and the action of da on u by

\[ dau = \text{action}(da, u) \]

In total, we can build the right hand side of the system to compute \(\frac{du}{dv}\) with the single line

\[ dL = \text{ff(L, v)} - \text{action(diff(a, v), u)} \]

or, using this function

\[ dL = \text{sensitivity}_\text{rhs}(a, u, L, v) \]

\u2014 ufl.formoperators.\texttt{set\_list\_item}(li, i, v)

\u2014 ufl.formoperators.\texttt{system}(form)

UFL form operator: Split a form into the left hand side and right hand side, see lhs and rhs.

\u2014 ufl.formoperators.\texttt{zero\_lists}(shape)

### 2.2.21 ufl.functions module

Types for representing function spaces.

\texttt{class ufl.functions.\texttt{AbstractFunctionSpace}}

\texttt{Bases: object}

\texttt{ufl\_sub\_spaces()}

\texttt{class ufl.functions.\texttt{FunctionSpace}(domain, element)}

\texttt{Bases: ufl.functions.\texttt{AbstractFunctionSpace}}

\texttt{ufl\_domain()}

\texttt{ufl\_element()}

\texttt{ufl\_sub\_spaces()}

\texttt{class ufl.functions.\texttt{MixedFunctionSpace}(*function\_spaces)}

\texttt{Bases: ufl.functions.\texttt{AbstractFunctionSpace}}

\texttt{ufl\_sub\_spaces()}

\texttt{class ufl.functions.\texttt{TensorProductFunctionSpace}(*function\_spaces)}

\texttt{Bases: ufl.functions.\texttt{AbstractFunctionSpace}}

\texttt{ufl\_sub\_spaces()}
### 2.2.22 ufl.geometry module

Types for representing symbolic expressions for geometric quantities.

#### class ufl.geometry.CellCoordinate(domain)

**Bases:** ufl.geometry.GeometricCellQuantity

UFL geometry representation: The coordinate in a reference cell.

In the context of expression integration, represents the reference cell coordinate of each quadrature point.

In the context of expression evaluation in a point in a cell, represents that point in the reference coordinate system of the cell.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'X'**

**ufl_shape**

#### class ufl.geometry.CellEdgeVectors(domain)

**Bases:** ufl.geometry.GeometricCellQuantity

UFL geometry representation: The vectors between reference cell vertices for each edge in cell.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'CEV'**

**ufl_shape**

#### class ufl.geometry.CellFacetJacobian(domain)

**Bases:** ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The Jacobian of the mapping from reference facet to reference cell coordinates.

\[
\text{CF}_j^i = \frac{dX_j}{dX_f}^i
\]

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'CFJ'**

**ufl_shape**

#### class ufl.geometry.CellFacetJacobianDeterminant(domain)

**Bases:** ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The pseudo-determinant of the CellFacetJacobian.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'detCFJ'**

#### class ufl.geometry.CellFacetJacobianInverse(domain)

**Bases:** ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The pseudo-inverse of the CellFacetJacobian.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'CFK'**
ufl_shape

class ufl.geometry.CellFacetOrigin(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    
    UFL geometry representation: The reference cell coordinate corresponding to origin of a reference facet.
    
    name = 'X0f'

ufl_shape

class ufl.geometry.CellNormal(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The upwards pointing normal vector of the current manifold cell.
    
    name = 'cell_normal'

ufl_shape

class ufl.geometry.CellOrientation(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The orientation (+1/-1) of the current cell.
    
    For non-manifold cells (tdim == gdim), this equals the sign of the Jacobian determinant, i.e. +1 if the physical cell is oriented the same way as the reference cell and -1 otherwise.
    
    For manifold cells of tdim==gdim-1 this is input data belonging to the mesh, used to distinguish between the sides of the manifold.
    
    name = 'cell_orientation'

ufl_shape

class ufl.geometry.CellOrigin(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The spatial coordinate corresponding to origin of a reference cell.
    
    is_cellwise_constant()
    
    name = 'x0'

ufl_shape

class ufl.geometry.CellVolume(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The volume of the cell.
    
    name = 'volume'

class ufl.geometry.Circumradius(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The circumradius of the cell.
    
    name = 'circumradius'

class ufl.geometry.FacetArea(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    
    UFL geometry representation: The area of the facet.
    
    name = 'facetarea'

class ufl.geometry.FacetCoordinate(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The coordinate in a reference cell of a facet.
In the context of expression integration over a facet, represents the reference facet coordinate of each quadrature point.
In the context of expression evaluation in a point on a facet, represents that point in the reference coordinate system of the facet.

```python
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
name = 'Xf'
```

```python
class ufl.geometry.FacetEdgeVectors(domain)
Bases: ufl.geometry.GeometricFacetQuantity
```
UFL geometry representation: The vectors between reference cell vertices for each edge in current facet.

```python
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
name = 'FEV'
```

```python
class ufl.geometry.FacetJacobian(domain)
Bases: ufl.geometry.GeometricFacetQuantity
```
UFL geometry representation: The Jacobian of the mapping from reference facet to spatial coordinates.

```
FJ_{ij} = \frac{dx_i}{dX_f j}
```
The FacetJacobian is the product of the Jacobian and CellFacetJacobian:

```
FJ = \frac{dx}{dXf} = \frac{dx}{dX} \frac{dX}{dXf} = J * CFJ
```

```python
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
name = 'FJ'
```

```python
class ufl.geometry.FacetJacobianDeterminant(domain)
Bases: ufl.geometry.GeometricFacetQuantity
```
UFL geometry representation: The pseudo-determinant of the FacetJacobian.

```python
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
name = 'detFJ'
```

```python
class ufl.geometry.FacetJacobianInverse(domain)
Bases: ufl.geometry.GeometricFacetQuantity
```
UFL geometry representation: The pseudo-inverse of the FacetJacobian.

```python
is_cellwise_constant()
```
Return whether this expression is spatially constant over each cell.

```python
name = 'FK'
```
class ufl.geometry.FacetNormal (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The outwards pointing normal vector of the current facet.
    is_cellwise_constant ()
        Return whether this expression is spatially constant over each cell.
    name = 'n'
    ufl_shape

class ufl.geometry.FacetOrientation (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The orientation (+1/-1) of the current facet relative to the reference cell.
    name = 'facet_orientation'

class ufl.geometry.FacetOrigin (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The spatial coordinate corresponding to origin of a reference facet.
    name = 'x0f'
    ufl_shape

class ufl.geometry.GeometricCellQuantity (domain)
    Bases: ufl.geometry.GeometricQuantity

class ufl.geometry.GeometricFacetQuantity (domain)
    Bases: ufl.geometry.GeometricQuantity

    is_cellwise_constant ()
        Return whether this expression is spatially constant over each cell (or over each facet for facet quantities).
    ufl_domains ()
    ufl_shape = ()

class ufl.geometry.Jacobian (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The Jacobian of the mapping from reference cell to spatial coordinates.
    J_ij = dx_i/dX_j
    is_cellwise_constant ()
        Return whether this expression is spatially constant over each cell.
    name = 'J'
    ufl_shape

class ufl.geometry.JacobianDeterminant (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The determinant of the Jacobian.
    Represents the signed determinant of a square Jacobian or the pseudo-determinant of a non-square Jacobian.
    is_cellwise_constant ()
        Return whether this expression is spatially constant over each cell.
name = 'detJ'

class ufl.geometry.JacobianInverse(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The inverse of the Jacobian.
    Represents the inverse of a square Jacobian or the pseudo-inverse of a non-square Jacobian.
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.

name = 'K'

ufl_shape

class ufl.geometry.MaxCellEdgeLength(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The maximum edge length of the cell.
    name = 'maxcelledgelength'

class ufl.geometry.MaxFacetEdgeLength(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The maximum edge length of the facet.
    name = 'maxfacetedgelength'

class ufl.geometry.MinCellEdgeLength(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The minimum edge length of the cell.
    name = 'mincelledgelength'

class ufl.geometry.MinFacetEdgeLength(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The minimum edge length of the facet.
    name = 'minfacetedgelength'

class ufl.geometry.QuadratureWeight(domain)
    Bases: ufl.geometry.GeometricQuantity
    UFL geometry representation: The current quadrature weight.
    Only used inside a quadrature context.
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.

name = 'weight'

class ufl.geometry.ReferenceCellVolume(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The volume of the reference cell.

name = 'reference_cell_volume'

class ufl.geometry.ReferenceFacetVolume(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The volume of the reference cell of the current facet.
name = 'reference_facet_volume'

class ufl.geometry.ReferenceNormal(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    
    UFL geometry representation: The outwards pointing normal vector of the current facet on the reference cell
    
    name = 'reference_normal'

ufl.shape

class ufl.geometry.SpatialCoordinate(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    
    UFL geometry representation: The coordinate in a domain.
    
    In the context of expression integration, represents the domain coordinate of each quadrature point.
    
    In the context of expression evaluation in a point, represents the value of that point.

    evaluate(x, mapping, component, index_values)
    
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.

    name = 'x'

ufl.shape

2.2.23 ufl.index_combination_utils module

Utilities for analysing and manipulating free index tuples

ufl.index_combination_utils. create_slice_indices(component, shape, fi)

ufl.index_combination_utils. merge_nonoverlapping_indices(a, b)
    Merge non-overlapping free indices into one representation.
    
    Example: C[i,j,r,s] = outer(A[i,s], B[j,r]) A, B -> (i,j,r,s), (idim,jdim,rdim,sdim)

ufl.index_combination_utils. merge_overlapping_indices(afi, afid, bfi, bfid)
    Merge overlapping free indices into one free and one repeated representation.
    
    Example: C[j,r] := A[i,j,k] * B[i,r,k] A, B -> (j,r), (jdim,rdim), (i,k), (idim,kdim)

ufl.index_combination_utils. merge_unique_indices(afi, afid, bfi, bfid)
    Merge two pairs of (index ids, index dimensions) sequences into one pair without duplicates.
    
    The id tuples afi, bfi are assumed already sorted by id. Given a list of (id, dim) tuples already sorted by id, return
    a unique list with duplicates removed. Also checks that the dimensions of duplicates are matching.

ufl.index_combination_utils. remove_indices(fi, fid, rfi)

ufl.index_combination_utils. unique_sorted_indices(indices)
    Given a list of (id, dim) tuples already sorted by id, return a unique list with duplicates removed. Also checks
    that the dimensions of duplicates are matching.

2.2.24 ufl.indexed module

This module defines the Indexed class.

class ufl.indexed.Indexed(expression, multiindex)
    Bases: ufl.core.operator.Operator

2.2. ufl package 147
evaluate \((x, mapping, component, index\_values, derivatives=())\)

ufl\_free\_indices

ufl\_index\_dimensions

ufl\_shape = ()

### 2.2.25 ufl.indexing module

This module defines the single index types and some internal index utilities.

### 2.2.26 ufl.indexsum module

This module defines the IndexSum class.

```python
class ufl.indexsum.IndexSum(summand, index)
    Bases: ufl.core.operator.Operator
dimension()
    evaluate \((x, mapping, component, index\_values)\)
    index()
    ufl\_free\_indices
    ufl\_index\_dimensions
    ufl\_shape
```

### 2.2.27 ufl.integral module

The Integral class.

```python
class ufl.integral.Integral(integrand, integral\_type, domain, subdomain\_id, metadata, subdomain\_data)
    Bases: object
    An integral over a single domain.
    domain()
    integral\_type()
        Return the domain type of this integral.
    integrand()
        Return the integrand expression, which is an Expr instance.
    metadata()
        Return the compiler metadata this integral has been annotated with.
    reconstruct(integrand=None, integral\_type=None, domain=None, subdomain\_id=None, metadata=None, subdomain\_data=None)
        Construct a new Integral object with some properties replaced with new values.
        Example: \(<a = \text{Integral instance}> b = a.\text{reconstruct}(\text{expand}\_\text{compounds}(a.\text{integrand}())) c = a.\text{reconstruct}(\text{metadata}={`\text{quadrature}\_\text{degree}`}:2))\)
    subdomain\_data()
        Return the domain data of this integral.
```
subdomain_id()
Return the subdomain id of this integral.

ufl_domain()
Return the integration domain of this integral.

### 2.2.28 ufl.log module

This module provides functions used by the UFL implementation to output messages. These may be redirected by the user of UFL.

```python
class ufl.log.Logger(name, exception_type=<type 'exceptions.Exception'>)
```

- **add_indent**(increment=1)
  Add to indentation level.

- **add_logfile**(filename=None, mode='a', level=10)

- **begin**(message)
  Begin task: write message and increase indentation level.

- **debug**(message)
  Write debug message.

- **deprecate**(message)
  Write deprecation message.

- **end**()
  End task: write a newline and decrease indentation level.

- **error**(message)
  Write error message and raise an exception.

- **get_handler**()
  Get handler for logging.

- **get_logfile_handler**(filename)

- **get_logger**()
  Return message logger.

- **info**(message)
  Write info message.

- **info_blue**(message)
  Write info message in blue.

- **info_green**(message)
  Write info message in green.

- **info_red**(message)
  Write info message in red.

- **log**(level, message)
  Write a log message on given log level

- **pop_level**()
  Pop log level from the level stack, reverting to before the last push_level.

- **push_level**(level)
  Push a log level on the level stack.
**set_handler** *(handler)*
Replace handler for logging. To add additional handlers instead of replacing the existing, use log.get_logger().addHandler(myhandler). See the logging module for more details.

**set_indent** *(level)*
Set indentation level.

**set_level** *(level)*
Set log level.

**set_prefix** *(prefix)*
Set prefix for log messages.

**warning** *(message)*
Write warning message.

**warning_blue** *(message)*
Write warning message in blue.

**warning_green** *(message)*
Write warning message in green.

**warning_red** *(message)*
Write warning message in red.

### 2.2.29 ufl.mathfunctions module

This module provides basic mathematical functions.

**class** ufl.mathfunctions.Acos *(argument)*
Bases: ufl.mathfunctions.MathFunction

**class** ufl.mathfunctions.Asin *(argument)*
Bases: ufl.mathfunctions.MathFunction

**class** ufl.mathfunctions.Atan *(argument)*
Bases: ufl.mathfunctions.MathFunction

**class** ufl.mathfunctions.Atan2 *(arg1, arg2)*
Bases: ufl.core.operator.Operator

**evaluate** *(x, mapping, component, index_values)*

**ufi_free_indices** = ()

**ufi_index_dimensions** = ()

**ufi_shape** = ()

**class** ufl.mathfunctions.BesselFunction *(name, classname, nu, argument)*
Bases: ufl.core.operator.Operator

Base class for all bessel functions

**evaluate** *(x, mapping, component, index_values)*

**ufi_free_indices** = ()

**ufi_index_dimensions** = ()

**ufi_shape** = ()

**class** ufl.mathfunctions.BesselI *(nu, argument)*
Bases: ufl.mathfunctions.BesselFunction
class ufl.mathfunctions.BesselJ(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.mathfunctions.BesselK(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.mathfunctions.BesselY(nu, argument)
    Bases: ufl.mathfunctions.BesselFunction

class ufl.mathfunctions.Cos(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Cosh(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Erf(argument)
    Bases: ufl.mathfunctions.MathFunction

    evaluate(x, mapping, component, index_values)

class ufl.mathfunctions.Exp(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Ln(argument)
    Bases: ufl.mathfunctions.MathFunction

    evaluate(x, mapping, component, index_values)

class ufl.mathfunctions.MathFunction(name, argument)
    Bases: ufl.core.operator.Operator

    Base class for all unary scalar math functions.

    evaluate(x, mapping, component, index_values)

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.mathfunctions.Sin(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Sinh(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Sqrt(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Tan(argument)
    Bases: ufl.mathfunctions.MathFunction

class ufl.mathfunctions.Tanh(argument)
    Bases: ufl.mathfunctions.MathFunction

2.2.30 ufl.measure module

The Measure class.

class ufl.measure.Measure(integral_type, domain=None, subdomain_id='everywhere', metadata=None, subdomain_data=None)
    Bases: object
domain()
integral_type()
    Return the domain type.
    Valid domain types are “cell”, “exterior_facet”, “interior_facet”, etc.
metadata()
    Return the integral metadata. This data is not interpreted by UFL. It is passed to the form compiler which can ignore it or use it to compile each integral of a form in a different way.
reconstruct(integral_type=None, subdomain_id=None, domain=None, metadata=None, subdomain_data=None)
    Construct a new Measure object with some properties replaced with new values.
    Example: <dm = Measure instance> b = dm.reconstruct(subdomain_id=2) c = dm.reconstruct(metadata={ "quadrature_degree": 3 })
    Used by the call operator, so this is equivalent: b = dm(2) c = dm(0, { "quadrature_degree": 3 })
subdomain_data()
    Return the integral subdomain_data. This data is not interpreted by UFL. Its intension is to give a context in which the domain id is interpreted.
subdomain_id()
    Return the domain id of this measure (integer).
ufl_domain()
    Return the domain associated with this measure.
    This may be None or a Domain object.
class ufl.measure.MeasureProduct(*measures)
    Bases: object
    Represents a product of measures.
    This is a notational intermediate object to handle the notation
    f*(dm1*dm2)
    This is work in progress and not functional. It needs support in other parts of ufl and the rest of the code generation chain.
sub_measures()
    Return submeasures.
class ufl.measure.MeasureSum(*measures)
    Bases: object
    Represents a sum of measures.
    This is a notational intermediate object to translate the notation
    f*(ds(1)+ds(3))
    into
    f*ds(1) + f*ds(3)
ufl.measure.as_integral_type(integral_type)
    Map short name to long name and require a valid one.
ufl.measure.integral_types()
    Return a tuple of all domain type strings.
ufl.measure.measure_names()
Return a tuple of all measure name strings.

ufl.measure.register_integral_type(integral_type, measure_name)

2.2.31 ufl.objects module
Utility objects for pretty syntax in user code.

2.2.32 ufl.operators module
This module extends the form language with free function operators, which are either already available as member
functions on UFL objects or defined as compound operators involving basic operations on the UFL objects.

ufl.operators.And(left, right)
UFL operator: A boolean expression (left and right) for use with conditional.

ufl.operators.Dn(f)
UFL operator: Take the directional derivative of f in the facet normal direction, Dn(f) := dot(grad(f), n).

ufl.operators.Dt(f)
UFL operator: <Not implemented yet!> The partial derivative of f with respect to time.

ufl.operators.Dx(f, *i)
UFL operator: Take the partial derivative of f with respect to spatial variable number i. Equivalent to f.dx(*i).

ufl.operators.Max(x, y)
UFL operator: Take the maximum of x and y.

ufl.operators.Min(x, y)
UFL operator: Take the minimum of x and y.

ufl.operators.Not(condition)
UFL operator: A boolean expression (not condition) for use with conditional.

ufl.operators.Or(left, right)
UFL operator: A boolean expression (left or right) for use with conditional.

ufl.operators.acos(f)
UFL operator: Take the inverse cosine of f.

ufl.operators.asin(f)
UFL operator: Take the inverse sin of f.

ufl.operators.atan(f)
UFL operator: Take the inverse tangent of f.

ufl.operators.atan_2(f1, f2)
UFL operator: Take the inverse tangent of f.

ufl.operators.avg(v)
UFL operator: Take the average of v across a facet.

ufl.operators.bessel_I(nu, f)
UFL operator: regular modified cylindrical Bessel function.

ufl.operators.bessel_J(nu, f)
UFL operator: cylindrical Bessel function of the first kind.

ufl.operators.bessel_K(nu, f)
UFL operator: irregular modified cylindrical Bessel function.
ufl.operators.bessel_Y(\(nu, f\))
   UFL operator: cylindrical Bessel function of the second kind.

ufl.operators.cell_avg(\(f\))
   UFL operator: Take the average of \(v\) over a cell.

ufl.operators.cofac(\(A\))
   UFL operator: Take the cofactor of \(A\).

ufl.operators.conditional(condition, true_value, false_value)
   UFL operator: A conditional expression, taking the value of true_value when condition evaluates to true and false_value otherwise.

ufl.operators.contraction(a, a_axes, b, b_axes)
   UFL operator: Take the contraction of \(a\) and \(b\) over given axes.

ufl.operators.cos(\(f\))
   UFL operator: Take the cosinus of \(f\).

ufl.operators.cosh(\(f\))
   UFL operator: Take the cosinus hyperbolicus of \(f\).

ufl.operators.cross(a, b)
   UFL operator: Take the cross product of \(a\) and \(b\).

ufl.operators.curl(\(f\))
   UFL operator: Take the curl of \(f\).

ufl.operators.det(\(A\))
   UFL operator: Take the determinant of \(A\).

ufl.operators.dev(\(A\))
   UFL operator: Take the deviatoric part of \(A\).

ufl.operators.diag(\(A\))
   UFL operator: Take the diagonal part of rank 2 tensor \(A\) _or_ make a diagonal rank 2 tensor from a rank 1 tensor.
   Always returns a rank 2 tensor. See also diag_vector.

ufl.operators.diag_vector(\(A\))
   UFL operator: Take the diagonal part of rank 2 tensor \(A\) and return as a vector.
   See also diag.

ufl.operators.diff(\(f, v\))
   UFL operator: Take the derivative of \(f\) with respect to the variable \(v\).
   If \(f\) is a form, diff is applied to each integrand.

ufl.operators.div(\(f\))
   UFL operator: Take the divergence of \(f\).
   This operator follows the div convention where
   \[
   \text{div}(v) = v[i].dx(i)
   \]
   \[
   \text{div}(T[:,i]) = T[:,i].dx(i)
   \]
   for vector expressions \(v\), and arbitrary rank tensor expressions \(T\).
   See also: nabla_div()

ufl.operators.dot(a, b)
   UFL operator: Take the dot product of \(a\) and \(b\).
ufl.operators.\texttt{elem\_div}(A, B)
  UFL operator: Take the elementwise division of the tensors A and B with the same shape.

ufl.operators.\texttt{elem\_mult}(A, B)
  UFL operator: Take the elementwise multiplication of the tensors A and B with the same shape.

ufl.operators.\texttt{elem\_op}(op, *args)
  UFL operator: Take the elementwise application of operator op on scalar values from one or more tensor arguments.

ufl.operators.\texttt{elem\_op\_items}(op\_ind, indices, *args)

ufl.operators.\texttt{elem\_pow}(A, B)
  UFL operator: Take the elementwise power of the tensors A and B with the same shape.

ufl.operators.\texttt{eq}(left, right)
  UFL operator: A boolean expression (left == right) for use with conditional.

ufl.operators.\texttt{erf}(f)
  UFL operator: Take the error function of f.

ufl.operators.\texttt{exp}(f)
  UFL operator: Take the exponential of f.

ufl.operators.\texttt{exterior\_derivative}(f)
  UFL operator: Take the exterior derivative of f.
  The exterior derivative uses the element family to determine whether id, grad, curl or div should be used.
  Note that this uses the ‘grad’ and ‘div’ operators, as opposed to ‘nabla\_grad’ and ‘nabla\_div’.

ufl.operators.\texttt{facet\_avg}(f)
  UFL operator: Take the average of v over a facet.

ufl.operators.\texttt{ge}(left, right)
  UFL operator: A boolean expression (left >= right) for use with conditional.

ufl.operators.\texttt{grad}(f)
  UFL operator: Take the gradient of f.
  This operator follows the grad convention where
  \[
  \text{grad}(s)[i] = s.dx(i) \\
  \text{grad}(v)[i, j] = v[i].dx(j) \\
  \text{grad}(T)[:, i] = T[::].dx(i)
  \]
  for scalar expressions s, vector expressions v, and arbitrary rank tensor expressions T.
  See also: \texttt{nabla\_grad()}

ufl.operators.\texttt{gt}(left, right)
  UFL operator: A boolean expression (left > right) for use with conditional.

ufl.operators.\texttt{inner}(a, b)
  UFL operator: Take the inner product of a and b.

ufl.operators.\texttt{inv}(A)
  UFL operator: Take the inverse of A.

ufl.operators.\texttt{jump}(v, n=None)
  UFL operator: Take the jump of v across a facet.

ufl.operators.\texttt{le}(left, right)
  UFL operator: A boolean expression (left <= right) for use with conditional.
ufl.operators.<code>ln</code>(<code>f</code>)
UFL operator: Take the natural logarithm of <code>f</code>.

ufl.operators.<code>lt</code>(<code>left</code>, <code>right</code>)
UFL operator: A boolean expression (<code>left</code> &lt; <code>right</code>) for use with conditional.

ufl.operators.<code>max_value</code>(<code>x</code>, <code>y</code>)
UFL operator: Take the maximum of <code>x</code> and <code>y</code>.

ufl.operators.<code>min_value</code>(<code>x</code>, <code>y</code>)
UFL operator: Take the minimum of <code>x</code> and <code>y</code>.

ufl.operators.<code>nabla_div</code>(<code>f</code>)
UFL operator: Take the divergence of <code>f</code>.
This operator follows the div convention where
\[
\nabla{\text{div}}(v) = v[i].dx(i)
\]
\[
\nabla{\text{div}}(T)[:] = T[i,:].dx(i)
\]
for vector expressions <code>v</code>, and arbitrary rank tensor expressions <code>T</code>.
See also: <code>div()</code>

ufl.operators.<code>nabla_grad</code>(<code>f</code>)
UFL operator: Take the gradient of <code>f</code>.
This operator follows the grad convention where
\[
\nabla{\text{grad}}(s)[i] = s.dx(i)
\]
\[
\nabla{\text{grad}}(v)[i,j] = v[j].dx(i)
\]
\[
\nabla{\text{grad}}(T)[i,:] = T[:].dx(i)
\]
for scalar expressions <code>s</code>, vector expressions <code>v</code>, and arbitrary rank tensor expressions <code>T</code>.
See also: <code>grad()</code>

ufl.operators.<code>ne</code>(<code>left</code>, <code>right</code>)
UFL operator: A boolean expression (<code>left</code> != <code>right</code>) for use with conditional.

ufl.operators.<code>outer</code> (*operands)
UFL operator: Take the outer product of two or more operands.

ufl.operators.<code>perp</code>(<code>v</code>)
UFL operator: Take the perp of <code>v</code>, i.e. (-v1, +v0).

ufl.operators.<code>rank</code>(<code>f</code>)
UFL operator: The rank of <code>f</code>.

ufl.operators.<code>rot</code>(<code>f</code>)
UFL operator: Take the curl of <code>f</code>.

ufl.operators.<code>shape</code>(<code>f</code>)
UFL operator: The shape of <code>f</code>.

ufl.operators.<code>sign</code>(<code>x</code>)
UFL operator: Take the sign (+1 or -1) of <code>x</code>.

ufl.operators.<code>sin</code>(<code>f</code>)
UFL operator: Take the sinus of <code>f</code>.

ufl.operators.<code>sinh</code>(<code>f</code>)
UFL operator: Take the sinus hyperbolicus of <code>f</code>.
ufl.operators.skew(A)
    UFL operator: Take the skew symmetric part of A.

ufl.operators.sqrt(f)
    UFL operator: Take the square root of f.

ufl.operators.sym(A)
    UFL operator: Take the symmetric part of A.

ufl.operators.tan(f)
    UFL operator: Take the tangent of f.

ufl.operators.tanh(f)
    UFL operator: Take the tangent hyperbolicus of f.

ufl.operators.tr(A)
    UFL operator: Take the trace of A.

ufl.operators.transpose(A)
    UFL operator: Take the transposed of tensor A.

ufl.operators.variable(e)
    UFL operator: Define a variable representing the given expression, see also diff().

2.2.33 ufl.permutation module

This module provides utility functions for computing permutations and generating index lists.

ufl.permutation.build_component_numbering(shape, symmetry)
    Build a numbering of components within the given value shape, taking into consideration a symmetry mapping
    which leaves the mapping noncontiguous. Returns a dict { component -> numbering } and an ordered list of
    components [ numbering -> component ]. The dict contains all components while the list only contains the ones
    not mapped by the symmetry mapping.

ufl.permutation.compute_indices(shape)
    Compute all index combinations for given shape

ufl.permutation.compute_indices2(shape)
    Compute all index combinations for given shape

ufl.permutation.compute_order_tuples(k, n)
    Compute all tuples of n integers such that the sum is k

ufl.permutation.compute_permutation_pairs(j, k)
    Compute all permutations of j + k elements from (0, j + k) in rising order within (0, j) and (j, j + k) respectively.

ufl.permutation.compute_permutations(k, n, skip=None)
    Compute all permutations of k elements from (0, n) in rising order. Any elements that are contained in the list
    skip are not included.

ufl.permutation.compute_sign(permutation)
    Compute sign by sorting.

2.2.34 ufl.precedence module

Precedence handling.

ufl.precedence.assign_precedences(precedence_list)
    Given a precedence list, assign ints to class._precedence.
ufl.precedence.build_precedence_list()

ufl.precedence.build_precedence_mapping(precedence_list)
  Given a precedence list, build a dict with class->int mappings. Utility function used by some external code.

ufl.precedence.parstr(child, parent, pre='(', post=')', format=<type ‘str’>)

2.2.35 ufl.protocols module

ufl.protocols.id_or_none(obj)
  Returns None if the object is None, obj.ufl_id() if available, or id(obj) if not.
  This allows external libraries to implement an alternative to id(obj) in the ufl_id() function, such that ufl can identify objects as the same without knowing about their types.

ufl.protocols.metadata_equal(a, b)

ufl.protocols.metadata_hashdata(md)

2.2.36 ufl.referencevalue module

Representation of the reference value of a function.

class ufl.referencevalue.ReferenceValue(f)
  Bases: ufl.core.operator.Operator

  Representation of the reference cell value of a form argument.
  evaluate(x, mapping, component, index_values, derivatives=())
    Get child from mapping and return the component asked for.

  ufl_free_indices = ()
  ufl_index_dimensions = ()
  ufl_shape

2.2.37 ufl.restriction module

Restriction operations.

class ufl.restriction.CellAvg(f)
  Bases: ufl.core.operator.Operator

  evaluate(x, mapping, component, index_values)
    Performs an approximate symbolic evaluation, since we dont have a cell.

  ufl_free_indices = ()
  ufl_index_dimensions = ()
  ufl_shape

class ufl.restriction.FacetAvg(f)
  Bases: ufl.core.operator.Operator

  evaluate(x, mapping, component, index_values)
    Performs an approximate symbolic evaluation, since we dont have a cell.

  ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape
class ufl.restriction.NegativeRestricted (f)
    Bases: ufl.restriction.Restricted
class ufl.restriction.PositiveRestricted (f)
    Bases: ufl.restriction.Restricted
class ufl.restriction.Restricted (f)
    Bases: ufl.core.operator.Operator
evaluate (x, mapping, component, index_values)
side ()
ufl_free_indices
ufl_index_dimensions
ufl_shape

2.2.38 ufl.sobolevspace module

This module defines a symbolic hierarchy of Sobolev spaces to enable symbolic reasoning about the spaces in which finite elements lie.
class ufl.sobolevspace.SobolevSpace (name, parents=None)
    Bases: object
    
    Symbolic representation of a Sobolev space. This implements a subset of the methods of a Python set so that finite elements and other Sobolev spaces can be tested for inclusion.

2.2.39 ufl.sorting module

This module contains a sorting rule for expr objects that is more robust w.r.t. argument numbering than using repr.
class ufl.sorting.ExprKey (x)
    Bases: object
    
x
ufl.sorting.cmp_expr (a, b)
ufl.sorting.cmp_expr2 (a, b)
    Sorting rule for Expr objects. NB! Do not use to compare for equality!
ufl.sorting.sorted_expr (seq)
ufl.sorting.sorted_expr_sum (seq)

2.2.40 ufl.split_functions module

Algorithm for splitting a Coefficient or Argument into subfunctions.
ufl.split_functions.split (v)
    UFL operator: If v is a Coefficient or Argument in a mixed space, returns a tuple with the function components corresponding to the subelements.
Compound tensor algebra operations.

class ufl.tensoralgebra.Cofactor(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape

class ufl.tensoralgebra.CompoundTensorOperator(operands)
    Bases: ufl.core.operator.Operator

class ufl.tensoralgebra.Cross(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = (3,)

class ufl.tensoralgebra.Determinant(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.tensoralgebra.Deviatoric(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.tensoralgebra.Dot(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.tensoralgebra.Inner(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.tensoralgebra.Inverse(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices = ()
    ufl_index_dimensions = ()
```python
class ufl.tensoralgebra.Outer(a, b)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.tensoralgebra.Skew(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.tensoralgebra.Sym(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.tensoralgebra.Trace(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.tensoralgebra.Transposed(A)
    Bases: ufl.tensoralgebra.CompoundTensorOperator
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

2.2.42 ufl.tensors module

Classes used to group scalar expressions into expressions with rank > 0.

class ufl.tensors.ComponentTensor(expression, indices)
    Bases: ufl.core.operator.Operator
    
    UFL operator type: Maps the free indices of a scalar valued expression to tensor axes.
    
    evaluate(x, mapping, component, index_values)
    indices()
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape
```

2.2. ufl package 161
class \texttt{ufl.tensors.ListTensor(*expressions)}
    Bases: \texttt{ufl.core.operator.Operator}

    UFL operator type: Wraps a list of expressions into a tensor valued expression of one higher rank.

    \texttt{evaluate(x, mapping, component, index\_values, derivatives=())}

\texttt{ufl\_free\_indices}

\texttt{ufl\_index\_dimensions}

\texttt{ufl\_shape}

\texttt{ufl.tensors.as\_matrix(expressions, indices=None)}

UFL operator: As \texttt{as\_tensor()}, but limited to rank 2 tensors.

\texttt{ufl.tensors.as\_scalar(expression)}

Given a scalar or tensor valued expression \(A\), returns either of the tuples:

\[
\begin{align*}
(a, b) &= (A, ()) \\
(a, b) &= (A[\text{indices}], \text{indices})
\end{align*}
\]

such that \(a\) is always a scalar valued expression.

\texttt{ufl.tensors.as\_scalars(*expressions)}

Given multiple scalar or tensor valued expressions \(A\), returns either of the tuples:

\[
\begin{align*}
(a, b) &= (A, ()) \\
(a, b) &= ([A[0][\text{indices}], ..., A[-1][\text{indices}]], \text{indices})
\end{align*}
\]

such that \(a\) is always a list of scalar valued expressions.

\texttt{ufl.tensors.as\_tensor(expressions, indices=None)}

UFL operator: Make a tensor valued expression.

This works in two different ways, by using indices or lists.

1) Returns \(A\) such that \(A[\text{indices}] = \text{expressions}\). If indices are provided, expressions must be a scalar valued expression with all the provided indices among its free indices. This operator will then map each of these indices to a tensor axis, thereby making a tensor valued expression from a scalar valued expression with free indices.

2) Returns \(A\) such that \(A[k,...] = \text{expressions[k]}\). If no indices are provided, expressions must be a list or tuple of expressions. The expressions can also consist of recursively nested lists to build higher rank tensors.

\texttt{ufl.tensors.as\_vector(expressions, index=None)}

UFL operator: As \texttt{as\_tensor()}, but limited to rank 1 tensors.

\texttt{ufl.tensors.dyad(d, *iota)}

TODO: Develop this concept, can e.g. write \(A[i,j]\*\text{dyad}(j,i)\) for the transpose.

\texttt{ufl.tensors.from\_numpy\_to\_lists(expressions)}

\texttt{ufl.tensors.numpy2nestedlists(arr)}

\texttt{ufl.tensors.relabel(A, indexmap)}

UFL operator: Relabel free indices of \(A\) with new indices, using the given mapping.

\texttt{ufl.tensors.unit\_indexed\_tensor(shape, component)}

\texttt{ufl.tensors.unit\_list(i, n)}

\texttt{ufl.tensors.unit\_list2(i, j, n)}

\texttt{ufl.tensors.unit\_matrices(d)}

UFL value: A tuple of constant unit matrices in all directions with dimension \(d\).
ufl.tensors.unit_matrix(i, j, d)
UFL value: A constant unit matrix in direction i,j with dimension d.

ufl.tensors.unit_vector(i, d)
UFL value: A constant unit vector in direction i with dimension d.

ufl.tensors.unit_vectors(d)
UFL value: A tuple of constant unit vectors in all directions with dimension d.

ufl.tensors.unwrap_list_tensor(lt)

2.2.43 ufl.variable module

Defines the Variable and Label classes, used to label expressions as variables for differentiation.

class ufl.variable.Label(count=None)
Bases: ufl.core.terminal.Terminal
count()
is_cellwise_constant()
ufldomains()
Return tuple of domains related to this terminal object.

ufldfree_indices
ufllindex_dimensions
uflshape

class ufl.variable.Variable(expression, label=None)
Bases: ufl.core.operator.Operator
A Variable is a representative for another expression.
It will be used by the end-user mainly for defining a quantity to differentiate w.r.t. using diff. Example:

\[
\begin{align*}
e & = <...> \\
e & = \text{variable}(e) \\
f & = \exp(e^2) \\
df & = \text{diff}(f, e)
\end{align*}
\]

evaluate(x, mapping, component, index_values)
expression()
label()
ufldomains()
ufllfree_indices =()
ufllindex_dimensions =()
uflshape

2.2.44 Module contents

The Unified Form Language is an embedded domain specific language for definition of variational forms intended for finite element discretization. More precisely, it defines a fixed interface for choosing finite element spaces and defining expressions for weak forms in a notation close to mathematical notation.
This Python module contains the language as well as algorithms to work with it.

- To import the language, type:
  ```python
  from ufl import *
  ```

- To import the underlying classes in an UFL expression tree is built from, type:
  ```python
  from ufl.classes import *
  ```

- Various algorithms for working with UFL expression trees can be found in:
  ```python
  from ufl.algorithms import *
  ```

The classes and algorithms are considered implementation details and should not be used in form definitions.

For more details on the language, see
http://www.fenicsproject.org

and
http://arxiv.org/abs/1211.4047

The development version can be found in the repository at
https://www.bitbucket.org/fenics-project/ufl

A very brief overview of the language contents follows:

- Cells:
  AbstractCell, Cell, TensorProductCell, OuterProductCell, vertex, interval, triangle, tetrahedron, quadrilateral, hexahedron

- Domains:
  AbstractDomain, Mesh, MeshView, TensorProductMesh

- Sobolev spaces:
  L2, H1, H2, HDiv, HCurl

- Elements:

- Function spaces:
  FunctionSpace

- Arguments:
  Argument, TestFunction, TrialFunction, Arguments, TestFunctions, TrialFunctions

- Coefficients:
Coefficient, Constant, VectorConstant, TensorConstant

- Splitting form arguments in mixed spaces:
  split

- Literal constants:
  Identity, PermutationSymbol

- Geometric quantities:
  SpatialCoordinate, FacetNormal, CellNormal, CellVolume, Circumradius, MinCellEdgeLength, MaxCellEdgeLength, FacetArea, MinFacetEdgeLength, MaxFacetEdgeLength, Jacobian, JacobianDeterminant, JacobianInverse

- Indices:
  Index, indices, i, j, k, l, p, q, r, s

- Scalar to tensor expression conversion:
  as_tensor, as_vector, as_matrix

- Unit vectors and matrices:
  unit_vector, unit_vectors, unit_matrix, unit_matrices

- Tensor algebra operators:
  outer, inner, dot, cross, perp, det, inv, cofac, transpose, tr, diag, diag_vector, dev, skew, sym

- Elementwise tensor operators:
  elem_mult, elem_div, elem_pow, elem_op

- Differential operators:
  variable, diff, grad, div, nabla_grad, nabla_div, Dx, Dn, curl, rot

- Nonlinear functions:
  max_value, min_value, abs, sign, sqrt, exp, ln, erf, cos, sin, tan, acos, asin, atan, atan_2, cosh, sinh, tanh, bessel_J, bessel_Y, bessel_I, bessel_K

- Discontinuous Galerkin operators: jump, avg, v(‘+’), v(‘-’), cell_avg, facet_avg

- Conditional operators:
eq, ne, le, ge, lt, gt,
<, >, <=, >=,
And, Or, Not,
conditional

- Integral measures:
  dx, ds, dS, dP,
dc, dC, dO, dI,
ds_b, ds_t, ds_t, ds_v, dS_h, dS_v

- Form transformations:
  rhs, lhs, system, functional,
  replace, replace_integral_domains,
  adjoint, action, energy_norm,
  sensitivity_rhs, derivative

ufl.product(sequence)
Return the product of all elements in a sequence.

exception ufl.UFLException
Bases: exceptions.Exception
Base class for UFL exceptions

ufl.as_cell(cell)
Convert any valid object to a Cell or return cell if it is already a Cell.
Allows an already valid cell, a known cellname string, or a tuple of cells for a product cell.

class ufl.AbstractCell(topological_dimension, geometric_dimension)
Bases: object
Representation of an abstract finite element cell with only the dimensions known.

generic_dimension()
Return the dimension of the space this cell is embedded in.

has_simplex_facets()
Return True if all the facets of this cell are simplex cells.

is_simplex()
Return True if this is a simplex cell.

topological_dimension()
Return the dimension of the topology of this cell.

class ufl.Cell(cellname, geometric_dimension=None)
Bases: ufl.cell.AbstractCell
Representation of a named finite element cell with known structure.

cellname()
Return the cellname of the cell.

has_simplex_facets()

is_simplex()

num_edges()
The number of cell edges.
num_facet_edges()  
The number of facet edges.

num_facets()  
The number of cell facets.

num_vertices()  
The number of cell vertices.

class ufl.TensorProductCell(cells)  
Bases: ufl.cell.AbstractCell

  has_simplex_facets()  
  Return True if all the facets of this cell are simplex cells.

  is_simplex()  
  Return True if this is a simplex cell.

  num_edges()  
  The number of cell edges.

  num_facets()  
  The number of cell facets.

  num_vertices()  
  The number of cell vertices.

  sub_cells()  
  Return list of cell factors.

class ufl.OuterProductCell(A, B, gdim=None)  
Bases: ufl.cell.AbstractCell

  Representation of a cell formed as the Cartesian product of two existing cells

  facet_horiz

  facet_vert

  has_simplex_facets()  
  Return True if all the facets of this cell are simplex cells.

  is_simplex()  
  Return True if this is a simplex cell.

  num_edges()  
  The number of cell edges.

  num_facets()  
  The number of cell facets.

  num_vertices()  
  The number of cell vertices.

ufl.as_domain(domain)  
Convert any valid object to an AbstractDomain type.

class ufl.AbstractDomain(topological_dimension, geometric_dimension)  
Bases: object

  Symbolic representation of a geometric domain with only a geometric and topological dimension.

  geometric_dimension()  
  Return the dimension of the space this domain is embedded in.
topological_dimension()
    Return the dimension of the topology of this domain.

class ufl.Mesh(coordinate_element, ufl_id=None, cargo=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    cell()
    coordinates()
    is_piecewise_linear_simplex_domain()
    ufl_cargo()
        Return carried object that will not be used by UFL.
    ufl_cell()
    ufl_coordinate_element()
    ufl_coordinates()
    ufl_id()
        Return the ufl_id of this object.

class ufl.MeshView(mesh, topological_dimension, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_id()
        Return the ufl_id of this object.
    ufl_mesh()

class ufl.TensorProductMesh(meshes, ufl_id=None)
    Bases: ufl.domain.AbstractDomain
    Symbolic representation of a mesh.
    is_piecewise_linear_simplex_domain()
    ufl_cell()
    ufl_coordinate_element()
    ufl_id()
        Return the ufl_id of this object.

class ufl.SpatialCoordinate(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The coordinate in a domain.
    In the context of expression integration, represents the domain coordinate of each quadrature point.
    In the context of expression evaluation in a point, represents the value of that point.
    evaluate(x, mapping, component, index_values)
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
name = 'x'

ufl_shape

class ufl.CellVolume (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The volume of the cell.
    name = 'volume'

class ufl.Circumradius (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The circumradius of the cell.
    name = 'circumradius'

class ufl.MinCellEdgeLength (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The minimum edge length of the cell.
    name = 'mincelledgelength'

class ufl.MaxCellEdgeLength (domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The maximum edge length of the cell.
    name = 'maxcelledgelength'

class ufl.FacetArea (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The area of the facet.
    name = 'facetarea'

class ufl.MinFacetEdgeLength (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The minimum edge length of the facet.
    name = 'minfacetedgelength'

class ufl.MaxFacetEdgeLength (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The maximum edge length of the facet.
    name = 'maxfacetedgelength'

class ufl.FacetNormal (domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The outwards pointing normal vector of the current facet.
    is_cellwise_constant ()
        Return whether this expression is spatially constant over each cell.
    name = 'n'

ufl_shape

class ufl.CellNormal (domain)
    Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The upwards pointing normal vector of the current manifold cell.

```python
name = 'cell_normal'
```

class `ufl.Jacobian`(*domain*)

Bases: `ufl.geometry.GeometricCellQuantity`

UFL geometry representation: The Jacobian of the mapping from reference cell to spatial coordinates.

\[ J_{ij} = \frac{dx_i}{dX_j} \]

```python
is_cellwise_constant()
```

Return whether this expression is spatially constant over each cell.

```python
name = 'J'
```

class `ufl.JacobianDeterminant`(*domain*)

Bases: `ufl.geometry.GeometricCellQuantity`

UFL geometry representation: The determinant of the Jacobian.

Represents the signed determinant of a square Jacobian or the pseudo-determinant of a non-square Jacobian.

```python
is_cellwise_constant()
```

Return whether this expression is spatially constant over each cell.

```python
name = 'detJ'
```

class `ufl.JacobianInverse`(*domain*)

Bases: `ufl.geometry.GeometricCellQuantity`

UFL geometry representation: The inverse of the Jacobian.

Represents the inverse of a square Jacobian or the pseudo-inverse of a non-square Jacobian.

```python
is_cellwise_constant()
```

Return whether this expression is spatially constant over each cell.

```python
name = 'K'
```

class `ufl.FiniteElementBase`(*family, cell, degree, quad_scheme, value_shape, reference_value_shape*)

Bases: `object`

Base class for all finite elements

```python
cell()
```

Return cell of finite element

```python
degree(*component=None*)
```

Return polynomial degree of finite element

```python
extract_component(*i*)
```

Recursively extract component index relative to a (simple) element and that element for given value component index

```python
extract_reference_component(*i*)
```

Recursively extract reference component index relative to a (simple) element and that element for given reference value component index
**extract_subelement_component** *(i)*
Extract direct subelement index and subelement relative component index for a given component index

**extract_subelement_reference_component** *(i)*
Extract direct subelement index and subelement relative reference component index for a given reference component index

**family**
Return finite element family

**is_cellwise_constant** *(component=None)*
Return whether the basis functions of this element is spatially constant over each cell.

**mapping**

**num_sub_elements**
Return number of sub elements

**quadrature_scheme**
Return quadrature scheme of finite element

**reference_value_shape**
Return the shape of the value space on the reference cell.

**reference_value_size**
Return the integer product of the reference value shape.

**sub_elements**
Return list of sub elements

**symmetry**
Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

**value_shape**
Return the shape of the value space on the global domain.

**value_size**
Return the integer product of the value shape.

---

**class ufl.FiniteElement** *(family, cell=None, degree=None, form_degree=None, quad_scheme=None)*

Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The basic finite element class for all simple finite elements

**mapping**

**shortstr**
Format as string for pretty printing.

**sobolev_space**

---

**class ufl.MixedElement** *(elements, **kwargs)*

Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

A finite element composed of a nested hierarchy of mixed or simple elements

**degree** *(component=None)*
Return polynomial degree of finite element

**extract_component** *(i)*
Recursively extract component index relative to a (simple) element and that element for given value component index
**extract_reference_component**(\(i\))
Recursively extract reference_component index relative to a (simple) element and that element for given
value reference_component index

**extract_subelement_component**(\(i\))
Extract direct subelement index and subelement relative component index for a given component index

**extract_subelement_reference_component**(\(i\))
Extract direct subelement index and subelement relative reference_component index for a given refer-
tence_component index

**is_cellwise_constant** *(\(component=None\))*
Return whether the basis functions of this element is spatially constant over each cell.

**mapping**()

**num_sub_elements**()
Return number of sub elements.

**reconstruct_from_elements** *(\(*elements*\))*
Reconstruct a mixed element from new subelements.

**shortstr**()
Format as string for pretty printing.

**sub_elements**()
Return list of sub elements.

**symmetry**()
Return the symmetry dict, which is a mapping \(c0 \rightarrow c1\) meaning that component \(c0\) is represented by
component \(c1\). A component is a tuple of one or more ints.

**class** ufl.VectorElement *(family, cell, degree, dim=None, form_degree=None, quad_scheme=None)*
Bases: ufl.finiteelement.mixedelement.MixedElement

A special case of a mixed finite element where all elements are equal

**shortstr**()
Format as string for pretty printing.

**class** ufl.TensorElement *(family, cell, degree, shape=None, symmetry=None, quad_scheme=None)*
Bases: ufl.finiteelement.mixedelement.MixedElement

A special case of a mixed finite element where all elements are equal

**extract_subelement_component**(\(i\))
Extract direct subelement index and subelement relative component index for a given component index

**flattened_sub_element_mapping**()

**mapping**()

**shortstr**()
Format as string for pretty printing.

**symmetry**()
Return the symmetry dict, which is a mapping \(c0 \rightarrow c1\) meaning that component \(c0\) is represented by
component \(c1\).

**class** ufl.EnrichedElement *(\(*elements*\))*
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The vector sum of two finite element spaces:

\[\text{EnrichedElement}(V, Q) = \{ v + q \mid v \in V, q \in Q \}.\]
is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

mapping()

shortstr()
Format as string for pretty printing.

class ufl.RestrictedElement (element, restriction_domain)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
Represents the restriction of a finite element to a type of cell entity.

element()

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

mapping()

num_restricted_sub_elements()
Return number of restricted sub elements.

num_sub_elements()
Return number of sub elements

restricted_sub_elements()
Return list of restricted sub elements.

restriction_domain()
Return the domain onto which the element is restricted.

shortstr()
Format as string for pretty printing.

sub_element()
Return the element which is restricted.

sub_elements()
Return list of sub elements

symmetry()
Return the symmetry dict, which is a mapping c0 -> c1 meaning that component c0 is represented by component c1.

class ufl.TensorProductElement (elements)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
The tensor product of d element spaces:

\[ V = V_0 \otimes V_1 \otimes \ldots \otimes V_d \]

Given bases \{phi_i\} for \(V_i\) for \(i = 1, \ldots, d\), \{ phi_0 * phi_1 * ... * phi_d \} forms a basis for \(V\).

mapping()

num_sub_elements()
Return number of subelements.

shortstr()
Short pretty-print.

sub_elements()
Return subelements (factors).
class `ufl.OuterProductElement` *(A, B, cell=None)*

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

The outer (tensor) product of 2 element spaces:

\[ V = A \otimes B \]

Given bases \( \phi_A, \phi_B \) for \( A, B \), \( \phi_A \ast \phi_B \) forms a basis for \( V \).

`mapping()`

`shortstr()`

Short pretty-print.

class `ufl.OuterProductVectorElement` *(A, B, cell=None, dim=None)*

Bases: `ufl.finiteelement.mixedelement.MixedElement`

A special case of a mixed finite element where all elements are equal OuterProductElements

`mapping()`

`shortstr()`

Format as string for pretty printing.

class `ufl.HDivElement` *(element)*

Bases: `ufl.finiteelement.outerproductelement.OuterProductElement`

A div-conforming version of an outer product element, assuming this makes mathematical sense.

`shortstr()`

class `ufl.HCurlElement` *(element)*

Bases: `ufl.finiteelement.outerproductelement.OuterProductElement`

A curl-conforming version of an outer product element, assuming this makes mathematical sense.

`shortstr()`

class `ufl.BrokenElement` *(element)*

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

The discontinuous version of an existing Finite Element space

`mapping()`

`shortstr()`

class `ufl.TraceElement` *(element)*

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

A finite element space: the trace of a given hdiv element. This is effectively a scalar-valued restriction which is non-zero only on cell facets.

`mapping()`

`shortstr()`

class `ufl.FacetElement` *(element)*

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

A version of an existing Finite Element space in which all dofs associated with the interior have been discarded

`mapping()`

`shortstr()`

174 Chapter 2. Manual and API Reference
class `ufl.InteriorElement` *(element)*

Bases: `ufl.finiteelement.finiteelementbase.FiniteElementBase`

A version of an existing Finite Element space in which only the dofs associated with the interior have been kept

```python
mapping()
shortstr()
```

`ufl.register_element` *(family, short_name, value_rank, sobolev_space, mapping, degree_range, cell_names)*

Register new finite element family

`ufl.show_elements()`

class `ufl.FunctionSpace` *(domain, element)*

Bases: `ufl.functionspace.AbstractFunctionSpace`

```python
ufl_domain()
ufl_element()
ufl_sub_spaces()
```

class `ufl.Argument` *(function_space, number, part=None)*

Bases: `ufl.core.terminal.FormArgument`

UFL value: Representation of an argument to a form.

```python
element()
is_cellwise_constant()
number()
part()
```

```python
ufl_domain()
ufl_domains()
```

Return tuple of domains related to this terminal object.

```python
ufl_element()
ufl_function_space()
```

Get the function space of this Argument.

```python
ufl_shape
```

`ufl.TestFunction` *(function_space, part=None)*

UFL value: Create a test function argument to a form.

`ufl.TrialFunction` *(function_space, part=None)*

UFL value: Create a trial function argument to a form.

`ufl.Arguments` *(function_space, number)*

UFL value: Create an Argument in a mixed space, and return a tuple with the function components corresponding to the subelements.

`ufl.TestFunctions` *(function_space)*

UFL value: Create a TestFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.
**ufl.TrialFunctions** *(function_space)*

UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

**class ufl.Coefficient** *(function_space, count=None)*

Bases: `ufl.core.terminal.FormArgument`

UFL form argument type: Representation of a form coefficient.

- **count()**
- **element()**
- **is_cellwise_constant()**
  
  Return whether this expression is spatially constant over each cell.

- **ufl_domain()**
  
  Shortcut to get the domain of the function space of this coefficient.

- **ufl_domains()**
  
  Return tuple of domains related to this terminal object.

- **ufl_element()**
  
  Shortcut to get the finite element of the function space of this coefficient.

- **ufl_function_space()**
  
  Get the function space of this coefficient.

**ufl.Coefficients** *(function_space)*

UFL value: Create a Coefficient in a mixed space, and return a tuple with the function components corresponding to the subelements.

**ufl.Constant** *(domain, count=None)*

UFL value: Represents a globally constant scalar valued coefficient.

**ufl.VectorConstant** *(domain, dim=None, count=None)*

UFL value: Represents a globally constant vector valued coefficient.

**ufl.TensorConstant** *(domain, shape=None, symmetry=None, count=None)*

UFL value: Represents a globally constant tensor valued coefficient.

**ufl.split(v)**

UFL operator: If v is a Coefficient or Argument in a mixed space, returns a tuple with the function components corresponding to the subelements.

**class ufl.PermutationSymbol** *(dim)*

Bases: `ufl.constantvalue.ConstantValue`

UFL literal type: Representation of a permutation symbol.

This is also known as the Levi-Civita symbol, antisymmetric symbol, or alternating symbol.

- **evaluate(x, mapping, component, index_values)**

**ufl_shape**

**class ufl.Identity** *(dim)*

Bases: `ufl.constantvalue.ConstantValue`

UFL literal type: Representation of an identity matrix.

- **evaluate(x, mapping, component, index_values)**

**ufl_shape**
ufl.zero(*shape)
UFL literal constant: Return a zero tensor with the given shape.

ufl.as_ufl(expression)
Converts expression to an Expr if possible.

class ufl.Index(count=None)
    Bases: ufl.core.multiindex.IndexBase
    UFL value: An index with no value assigned.
    Used to represent free indices in Einstein indexing notation.
    count()

ufl.indices(n)
UFL value: Return a tuple of n new Index objects.

ufl.as_tensor(expressions, indices=None)
UFL operator: Make a tensor valued expression.
This works in two different ways, by using indices or lists.
1) Returns A such that A[indices] = expressions. If indices are provided, expressions must be a scalar valued expression with all the provided indices among its free indices. This operator will then map each of these indices to a tensor axis, thereby making a tensor valued expression from a scalar valued expression with free indices.
2) Returns A such that A[k,...] = expressions[k]. If no indices are provided, expressions must be a list or tuple of expressions. The expressions can also consist of recursively nested lists to build higher rank tensors.

ufl.as_vector(expressions, index=None)
UFL operator: As as_tensor(), but limited to rank 1 tensors.

ufl.as_matrix(expressions, indices=None)
UFL operator: As as_tensor(), but limited to rank 2 tensors.

ufl.relabel(A, indexmap)
UFL operator: Relabel free indices of A with new indices, using the given mapping.

ufl.unit_vector(i, d)
UFL value: A constant unit vector in direction i with dimension d.

ufl.unit_vectors(d)
UFL value: A tuple of constant unit vectors in all directions with dimension d.

ufl.unit_matrix(i, j, d)
UFL value: A constant unit matrix in direction i,j with dimension d.

ufl.unit_matrices(d)
UFL value: A tuple of constant unit matrices in all directions with dimension d.

ufl.rank(f)
UFL operator: The rank of f.

ufl.shape(f)
UFL operator: The shape of f.

ufl.outer(*operands)
UFL operator: Take the outer product of two or more operands.

ufl.inner(a, b)
UFL operator: Take the inner product of a and b.

ufl.dot(a, b)
UFL operator: Take the dot product of a and b.
ufl\texttt{.cross}(a, b)
\hspace{0.3cm} \text{UFL operator: Take the cross product of a and b.}

ufl\texttt{.perp}(v)
\hspace{0.3cm} \text{UFL operator: Take the perp of v, i.e. (-v1, +v0).}

ufl\texttt{.det}(A)
\hspace{0.3cm} \text{UFL operator: Take the determinant of A.}

ufl\texttt{.inv}(A)
\hspace{0.3cm} \text{UFL operator: Take the inverse of A.}

ufl\texttt{.cofac}(A)
\hspace{0.3cm} \text{UFL operator: Take the cofactor of A.}

ufl\texttt{.transpose}(A)
\hspace{0.3cm} \text{UFL operator: Take the transposed of tensor A.}

ufl\texttt{.tr}(A)
\hspace{0.3cm} \text{UFL operator: Take the trace of A.}

ufl\texttt{.diag}(A)
\hspace{0.3cm} \text{UFL operator: Take the diagonal part of rank 2 tensor A \_or\_ make a diagonal rank 2 tensor from a rank 1 tensor.}
\hspace{0.3cm} \text{Always returns a rank 2 tensor. See also diag\_vector.}

ufl\texttt{.diag\_vector}(A)
\hspace{0.3cm} \text{UFL operator: Take the diagonal part of rank 2 tensor A and return as a vector.}
\hspace{0.3cm} \text{See also diag.}

ufl\texttt{.dev}(A)
\hspace{0.3cm} \text{UFL operator: Take the deviatoric part of A.}

ufl\texttt{.skew}(A)
\hspace{0.3cm} \text{UFL operator: Take the skew symmetric part of A.}

ufl\texttt{.sym}(A)
\hspace{0.3cm} \text{UFL operator: Take the symmetric part of A.}

ufl\texttt{.sqrt}(f)
\hspace{0.3cm} \text{UFL operator: Take the square root of f.}

ufl\texttt{.exp}(f)
\hspace{0.3cm} \text{UFL operator: Take the exponential of f.}

ufl\texttt{.ln}(f)
\hspace{0.3cm} \text{UFL operator: Take the natural logarithm of f.}

ufl\texttt{.erf}(f)
\hspace{0.3cm} \text{UFL operator: Take the error function of f.}

ufl\texttt{.cos}(f)
\hspace{0.3cm} \text{UFL operator: Take the cosinus of f.}

ufl\texttt{.sin}(f)
\hspace{0.3cm} \text{UFL operator: Take the sinus of f.}

ufl\texttt{.tan}(f)
\hspace{0.3cm} \text{UFL operator: Take the tangent of f.}

ufl\texttt{.acos}(f)
\hspace{0.3cm} \text{UFL operator: Take the inverse cosinus of f.}
ufl.asin(f)
   UFL operator: Take the inverse sinus of f.

ufl.atan(f)
   UFL operator: Take the inverse tangent of f.

ufl.atan_2(f1, f2)
   UFL operator: Take the inverse tangent of f.

ufl.cosh(f)
   UFL operator: Take the cosinus hyperbolicus of f.

ufl.sinh(f)
   UFL operator: Take the sinus hyperbolicus of f.

ufl.tanh(f)
   UFL operator: Take the tangent hyperbolicus of f.

ufl.bessel_J(nu, f)
   UFL operator: cylindrical Bessel function of the first kind.

ufl.bessel_Y(nu, f)
   UFL operator: cylindrical Bessel function of the second kind.

ufl.bessel_I(nu, f)
   UFL operator: regular modified cylindrical Bessel function.

ufl.bessel_K(nu, f)
   UFL operator: irregular modified cylindrical Bessel function.

ufl.eq(left, right)
   UFL operator: A boolean expresion (left == right) for use with conditional.

ufl.ne(left, right)
   UFL operator: A boolean expresion (left != right) for use with conditional.

ufl.ue(left, right)
   UFL operator: A boolean expresion (left <= right) for use with conditional.

ufl.ge(left, right)
   UFL operator: A boolean expresion (left >= right) for use with conditional.

ufl.it(left, right)
   UFL operator: A boolean expresion (left < right) for use with conditional.

ufl.gt(left, right)
   UFL operator: A boolean expresion (left > right) for use with conditional.

ufl.And(left, right)
   UFL operator: A boolean expresion (left and right) for use with conditional.

ufl.Or(left, right)
   UFL operator: A boolean expresion (left or right) for use with conditional.

ufl.Not(condition)
   UFL operator: A boolean expresion (not condition) for use with conditional.

ufl.conditional(condition, true_value, false_value)
   UFL operator: A conditional expression, taking the value of true_value when condition evaluates to true and false_value otherwise.

ufl.sign(x)
   UFL operator: Take the sign (+1 or -1) of x.
ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_max\_value}\ (x, y)
UFL operator: Take the maximum of x and y.

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_min\_value}\ (x, y)
UFL operator: Take the minimum of x and y.

ufl.\texttt{Max}\ (x, y)
UFL operator: Take the maximum of x and y.

ufl.\texttt{Min}\ (x, y)
UFL operator: Take the minimum of x and y.

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_variable}\ (e)
UFL operator: Define a variable representing the given expression, see also diff().

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_diff}\ (f, v)
UFL operator: Take the derivative of f with respect to the variable v.
If f is a form, diff is applied to each integrand.

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Dx}\ (f, *i)
UFL operator: Take the partial derivative of f with respect to spatial variable number i. Equivalent to f.dx(*i).

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_grad}\ (f)
UFL operator: Take the gradient of f.
This operator follows the grad convention where
\[
\text{grad}(s)[i] = s.dx(i) \\
\text{grad}(v)[i,j] = v[i].dx(j) \\
\text{grad}(T)[:,i] = T[:,].dx(i)
\]
for scalar expressions s, vector expressions v, and arbitrary rank tensor expressions T.
See also: \texttt{nabla\_grad()}

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_div}\ (f)
UFL operator: Take the divergence of f.
This operator follows the div convention where
\[
\text{div}(v) = v[i].dx(i) \\
\text{div}(T)[:] = T[:,i].dx(i)
\]
for vector expressions v, and arbitrary rank tensor expressions T.
See also: \texttt{nabla\_div()}

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_curl}\ (f)
UFL operator: Take the curl of f.

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_rot}\ (f)
UFL operator: Take the curl of f.

ufl.\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_nabla\_grad}\ (f)
UFL operator: Take the gradient of f.
This operator follows the grad convention where
\[
\text{nabla\_grad}(s)[i] = s.dx(i) \\
\text{nabla\_grad}(v)[i,j] = v[j].dx(i) \\
\text{nabla\_grad}(T)[i,:] = T[:,].dx(i)
\]
for scalar expressions $s$, vector expressions $v$, and arbitrary rank tensor expressions $T$.

See also: $\textit{grad()}$

\texttt{ufl.nabla\_div}(f)
UFL operator: Take the divergence of $f$.

This operator follows the $\text{div}$ convention where

\[
\text{nabla\_div}(v) = v[i].dx(i) \\
\text{nabla\_div}(T)[i:] = T[i,:].dx(i)
\]

for vector expressions $v$, and arbitrary rank tensor expressions $T$.

See also: $\textit{div()}$

\texttt{ufl.Dn}(f)
UFL operator: Take the directional derivative of $f$ in the facet normal direction, $D_n(f) := \text{dot}(\text{grad}(f), n)$.

\texttt{ufl.exterior\_derivative}(f)
UFL operator: Take the exterior derivative of $f$.

The exterior derivative uses the element family to determine whether $\text{id}$, $\text{grad}$, $\text{curl}$ or $\text{div}$ should be used.

Note that this uses the ‘grad’ and ‘div’ operators, as opposed to ‘nabla\_grad’ and ‘nabla\_div’.

\texttt{ufl.jump}(v, n=None)
UFL operator: Take the jump of $v$ across a facet.

\texttt{ufl.avg}(v)
UFL operator: Take the average of $v$ across a facet.

\texttt{ufl.cell\_avg}(f)
UFL operator: Take the average of $v$ over a cell.

\texttt{ufl.facet\_avg}(f)
UFL operator: Take the average of $v$ over a facet.

\texttt{ufl.elem\_mult}(A, B)
UFL operator: Take the elementwise multiplication of the tensors $A$ and $B$ with the same shape.

\texttt{ufl.elem\_div}(A, B)
UFL operator: Take the elementwise division of the tensors $A$ and $B$ with the same shape.

\texttt{ufl.elem\_pow}(A, B)
UFL operator: Take the elementwise power of the tensors $A$ and $B$ with the same shape.

\texttt{ufl.elem\_op}(op, *args)
UFL operator: Take the elementwise application of operator $op$ on scalar values from one or more tensor arguments.

class \texttt{ufl.Form}(integrals)
Bases: object

Description of a weak form consisting of a sum of integrals over subdomains.

\texttt{arguments}()
Return all Argument objects found in form.

\texttt{cell}()

\texttt{coefficient\_numbering}()
Return a contiguous numbering of coefficients in a mapping { coefficient: number }.  

2.2. \texttt{ufl} package
coefficients()
    Return all Coefficient objects found in form.

domain()

domain_numbering()
    Return a contiguous numbering of domains in a mapping { domain: number }.

domains()

empty()
    Returns whether the form has no integrals.

equals(other)
    Evaluate ‘bool(lhs_form == rhs_form)’.

gеомеtric dimension()
    Return the geometric dimension shared by all domains and functions in this form.

integrals()
    Return a sequence of all integrals in form.

integrals_by_type(integral_type)
    Return a sequence of all integrals with a particular domain type.

max_subdomain_ids()
    Returns a mapping on the form { domain: { integral_type: max_subdomain_id } }.

signature()
    Signature for use with jit cache (independent of incidental numbering of indices etc.)

subdomain_data()
    Returns a mapping on the form { domain: { integral_type: subdomain_data } }.

ufl_cell()
    Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()
    Return the single geometric integration domain occurring in the form.
    Fails if multiple domains are found.
    NB! This does not include domains of coefficients defined on other meshes, look at form data for that
    additional information.

ufl_domains()
    Return the geometric integration domains occurring in the form.
    NB! This does not include domains of coefficients defined on other meshes.
    The return type is a tuple even if only a single domain exists.

x_repr_latex_

x_repr_png_

class ufl.Integral(integrand, integral_type, domain, subdomain_id, metadata, subdomain_data)
    Bases: object
    An integral over a single domain.

domain()

integral_type()
    Return the domain type of this integral.
integrand()
Return the integrand expression, which is an Expr instance.

metadata()
Return the compiler metadata this integral has been annotated with.

reconstruct(integrand=None, integral_type=None, domain=None, subdomain_id=None, metadata=None, subdomain_data=None)
Construct a new Integral object with some properties replaced with new values.

Example: `<a = Integral instance> b = a.reconstruct(expand_compounds(a.integrand())) c = a.reconstruct(metadata={'quadrature_degree':2})`

subdomain_data()
Return the domain data of this integral.

subdomain_id()
Return the subdomain id of this integral.

ufl_domain()
Return the integration domain of this integral.

class ufl.Measure(integral_type, domain=None, subdomain_id='everywhere', metadata=None, subdomain_data=None)
Bases: object

domain()

integral_type()
Return the domain type.
Valid domain types are “cell”, “exterior_facet”, “interior_facet”, etc.

metadata()
Return the integral metadata. This data is not interpreted by UFL. It is passed to the form compiler which can ignore it or use it to compile each integral of a form in a different way.

reconstruct(integral_type=None, subdomain_id=None, domain=None, metadata=None, subdomain_data=None)
Construct a new Measure object with some properties replaced with new values.

Example: `<dm = Measure instance> b = dm.reconstruct(subdomain_id=2) c = dm.reconstruct(metadata={ “quadrature_degree”: 3 })`

Used by the call operator, so this is equivalent: `b = dm(2) c = dm(0, { “quadrature_degree”: 3 })`

subdomain_data()
Return the integral subdomain data. This data is not interpreted by UFL. Its intension is to give a context in which the domain id is interpreted.

subdomain_id()
Return the domain id of this measure (integer).

ufl_domain()
Return the domain associated with this measure.
This may be None or a Domain object.

ufl.register_integral_type(integral_type, measure_name)
ufl.integral_types()
Return a tuple of all domain type strings.
ufl.replace(e, mapping)
Replace terminal objects in expression.
@param e: An Expr or Form.

@param mapping: A dict with from:to replacements to perform.

ufl.replace_integral_domains(form, common_domain)

Given a form and a domain, assign a common integration domain to all integrals.

Does not modify the input form (Form should always be immutable). This is to support ill formed forms with no domain specified, some times occuring in pydolfin, e.g. assemble(1*dx, mesh=mesh).

ufl.derivative(form, coefficient, argument=None, coefficient_derivatives=None)

UFL form operator: Compute the Gateaux derivative of form w.r.t. coefficient in direction of argument.

If the argument is omitted, a new Argument is created in the same space as the coefficient, with argument number one higher than the highest one in the form.

The resulting form has one additional Argument in the same finite element space as the coefficient.

A tuple of Coefficients may be provided in place of a single Coefficient, in which case the new Argument argument is based on a MixedElement created from this tuple.

An indexed Coefficient from a mixed space may be provided, in which case the argument should be in the corresponding subspace of the coefficient space.

If provided, coefficient_derivatives should be a mapping from Coefficient instances to their derivatives w.r.t. ‘coefficient’.

ufl.action(form, coefficient=None)

UFL form operator: Given a bilinear form, return a linear form with an additional coefficient, representing the action of the form on the coefficient. This can be used for matrix-free methods.

ufl.energy_norm(form, coefficient=None)

UFL form operator: Given a bilinear form a and a coefficient f, return the functional a(f,f).

ufl.rhs(form)

UFL form operator: Given a combined bilinear and linear form, extract the right hand side (negated linear form part).

Example:

\[ a = u*v*\text{dx} + f*v*\text{dx} \]
\[ L = \text{rhs}(a) \rightarrow -f*v*\text{dx} \]

ufl.lhs(form)

UFL form operator: Given a combined bilinear and linear form, extract the left hand side (bilinear form part).

Example:

\[ a = u*v*\text{dx} + f*v*\text{dx} \]
\[ a = \text{lhs}(a) \rightarrow u*v*\text{dx} \]

ufl.system(form)

UFL form operator: Split a form into the left hand side and right hand side, see lhs and rhs.

ufl.functional(form)

UFL form operator: Extract the functional part of form.

ufl.adjoint(form, reordered_arguments=None)

UFL form operator: Given a combined bilinear form, compute the adjoint form by changing the ordering (count) of the test and trial functions.

By default, new Argument objects will be created with opposite ordering. However, if the adjoint form is to be added to other forms later, their arguments must match. In that case, the user must provide a tuple reordered_arguments=(u2,v2).
UFL form operator: Compute the right hand side for a sensitivity calculation system.

The derivation behind this computation is as follows. Assume \( a, L \) to be bilinear and linear forms corresponding to the assembled linear system

\[ Ax = b. \]

Where \( x \) is the vector of the discrete function corresponding to \( u \). Let \( v \) be some scalar variable this equation depends on. Then we can write

\[ 0 = \frac{d}{dv}[Ax-b] = \frac{dA}{dv} x + A \frac{dx}{dv} - \frac{db}{dv}, A \frac{dx}{dv} = \frac{db}{dv} - \frac{dA}{dv} x, \]

and solve this system for \( \frac{dx}{dv} \), using the same bilinear form \( a \) and matrix \( A \) from the original system. Assume the forms are written

\[ v = \text{variable}(v\_\text{expression}) \quad L = IL(v)\*dx \quad a = Ia(v)\*dx \]

where \( IL \) and \( Ia \) are integrand expressions. Define a Coefficient \( u \) representing the solution to the equations. Then we can compute \( \frac{db}{dv} \) and \( \frac{dA}{dv} \) from the forms

\[ da = \text{diff}(a, v) \quad dL = \text{diff}(L, v) \]

and the action of \( da \) on \( u \) by

\[ dau = \text{action}(da, u) \]

In total, we can build the right hand side of the system to compute \( \frac{du}{dv} \) with the single line

\[ dL = \text{diff}(L, v) - \text{action}(\text{diff}(a, v), u) \]

or, using this function

\[ dL = \text{sensitivity\_rhs}(a, u, L, v) \]

## 2.3 Release notes

### 2.3.1 Changes in the next release of UFL

- **Deprecate** `.cell()`, `.domain()`, `.element()` in favour of `.ufl_cell()`, `.ufl_domain()`, `.ufl_element()`, in multiple classes, to allow closer integration with DOLFIN.
- **Remove** deprecated properties `cell.{d,x,n,volume,circumradius,facet_area}`.
- **Remove** ancient `form2ufl` script
- **Large reworking** of symbolic geometry pipeline
- **Implement** symbolic Piola mappings

### 2.3.2 Changes in UFL 1.6.0

UFL 1.6.0 was released on 2015-07-28

- **Change** approach to attaching `__hash__` implementation to accomodate Python 3
- **Implement** new non-recursive traversal based hash computation
- **Allow** `derivative(M, ListTensor(<scalars>), ...)` just like list/tuple works
- **Add traits** `is_in_reference_frame, is_restriction, is_evaluation, is_differential`
- **Add** missing linear operators to `ArgumentDependencyExtractor`
• Add _ufl_is_literal_ type trait
• Add _ufl_is_terminal_modifier_ type trait and Expr._ufl_terminal_modifiers_ list
• Add new types ReferenceDiv and ReferenceCurl
• Outer product element support in degree estimation
• Add TraceElement, InteriorElement, FacetElement, BrokenElement
• Add OuterProductCell to valid Real elements
• Add _cache member to form for use by external frameworks
• Add Sobolev space HEin
• Add measures dI, dO, dC for interface, overlap, cutcell
• Remove Measure constants
• Remove cell2D and cell3D
• Implement reference_value in apply_restrictions
• Rename point integral to vertex integral and kept *dP syntax
• Replace lambda functions in ufl_type with named functions for nicer stack traces
• Minor bugfixes, removal of unused code and cleanups
  • modindex
  • genindex
  • search
ufl, 163
ufl.algebra, 97
ufl.algorithms, 75
ufl.algorithms.ad, 46
ufl.algorithms.analysis, 46
ufl.algorithms.apply_algebra_lowering, 47
ufl.algorithms.apply_derivatives, 48
ufl.algorithms.apply_function_pullbacks, 52
ufl.algorithms.apply_geometry_lowering, 52
ufl.algorithms.apply_integral_scaling, 53
ufl.algorithms.apply_restrictions, 53
ufl.algorithms.argument_dependencies, 55
ufl.algorithms.change_to_reference, 57
ufl.algorithms.check_arities, 58
ufl.algorithms.check_restrictions, 59
ufl.algorithms.checks, 59
ufl.algorithms.compute_form_data, 59
ufl.algorithms.domain_analysis, 60
ufl.algorithms.elementtransformations, 61
ufl.algorithms.estimate_degrees, 61
ufl.algorithms.expand_compounds, 64
ufl.algorithms.expand_indices, 64
ufl.algorithms.formdata, 64
ufl.algorithms.formfiles, 65
ufl.algorithms.formtransformations, 65
ufl.algorithms.forward_ad, 67
ufl.algorithms.map_integrands, 70
ufl.algorithms.multifunction, 71
ufl.algorithms.pdiffs, 71
ufl.algorithms.predicates, 72
ufl.algorithms.renumbering, 72
ufl.algorithms.replace, 73
ufl.algorithms.signature, 73
A
Abs (class in ufl.algebra), 97
Abs (class in ufl.classes), 112
abs() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset
method), 49
abs() (ufl.algorithms.estimate_degrees.SumDegreeEstimator
method), 61
abs() (ufl.algorithms.forward_ad.ForwardAD method), 67
abs() (ufl.algorithms.pdiffs.PartialDerivativeComputer
method), 71
abs() (ufl.formatting.ufl2latex.Expression2LatexHandler
method), 90
AbstractCell (class in ufl), 166
AbstractCell (class in ufl.cell), 99
AbstractCell (class in ufl.classes), 120
AbstractDomain (class in ufl), 167
AbstractDomain (class in ufl.classes), 125
AbstractDomain (class in ufl.domain), 136
AbstractFunctionSpace (class in ufl.classes), 126
AbstractFunctionSpace (class in ufl.functionspace), 141
accumulate_integrands_with_same_metadata() (in module ufl.algorithms.domain_analysis), 60
Acos (class in ufl.classes), 119
Acos (class in ufl.mathfunctions), 150
acos() (in module ufl), 178
acos() (in module ufl.operators), 153
acos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset
method), 49
acos() (ufl.algorithms.forward_ad.ForwardAD method), 67
acos() (ufl.algorithms.pdiffs.PartialDerivativeComputer
method), 71
acos() (ufl.formatting.ufl2latex.Expression2LatexHandler
method), 90
action() (in module ufl), 184
action() (in module ufl.operators), 140
add_indent() (ufl.log.Logger method), 149
add_logfile() (ufl.log.Logger method), 149
adj_expr() (in module ufl.compound_expressions), 130
adj_expr_2x2() (in module ufl.compound_expressions), 131
adj_expr_3x3() (in module ufl.compound_expressions), 131
adj_expr_4x4() (in module ufl.compound_expressions), 131
adjoint() (in module ufl), 184
adjoint() (in module ufl.operators), 140
affine_mesh() (in module ufl.domain), 137
align() (in module ufl.formatting.latextools), 89
alternative_dot() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra
method), 47
alternative_inner() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra
method), 47
always_reconstruct() (ufl.algorithms.transformer.Transformer
method), 74
analyse_key() (in module ufl.exproperators), 138
And() (in module ufl), 179
And() (in module ufl.operators), 153
and_condition() (ufl.formatting.ufl2latex.Expression2LatexHandler
method), 90
and_tuples() (in module ufl.utils.sequences), 95
AndCondition (class in ufl.classes), 117
AndCondition (class in ufl.conditional), 131
apply_algebra_lowering() (in module ufl.algorithms.apply_algebra_lowering), 48
apply_derivatives() (in module ufl.algorithms.apply_derivatives), 52
apply_function_pullbacks() (in module ufl.algorithms.apply_function_pullbacks), 52
apply_geometry_lowering() (in module ufl.algorithms.apply_geometry_lowering), 53
apply_integral_scaling() (in module ufl.algorithms.apply_integral_scaling), 53
apply_nested_forward_ad() (in module ufl.algorithms.forward_ad), 70
apply_restrictions() (in module ufl.algorithms.apply_restrictions), 55
apply_single_function_pullbacks() (in module ufl.algorithms.apply_function_pullbacks), 52
apply_transformer() (in module ufl.algorithms.transformer), 75
Argument (class in ufl), 175
Argument (class in ufl.argument), 98
Argument (class in ufl.classes), 110
argument() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 48
argument() (ufl.algorithms.apply_derivatives.GradRuleset method), 50
argument() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 51
argument() (ufl.algorithms.apply_derivatives.VariableRuleset method), 51
argument() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 53
argument() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtracter method), 55
argument() (ufl.algorithms.check_arities.ArityChecker method), 58
argument() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 61
argument() (ufl.algorithms.formtransformations.PartExtracter method), 65
argument() (ufl.algorithms.forward_ad.GradAD method), 69
argument() (ufl.formatting.graph.StringDependencyDefiner method), 88
argument() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 90
ArgumentDependencyExtracter (class in ufl.algorithms.argument_dependencies), 55
Arguments() (in module ufl), 175
Arguments() (in module ufl.argument), 98
arguments() (ufl.classes.Form method), 129
arguments() (ufl.Form method), 181
arguments() (ufl.form.Form method), 138
ArityChecker (class in ufl.algorithms.check_arities), 58
ArityMismatch, 59
as_cell() (in module ufl), 166
as_cell() (in module ufl.cell), 100
as_domain() (in module ufl), 167
as_domain() (in module ufl.domain), 137
as_form() (in module ufl.form), 139
as_integral_type() (in module ufl.measure), 152
as_matrix() (in module ufl), 177
as_matrix() (in module ufl.tensors), 162
as_multi_index() (in module ufl.core.multiindex), 78
as_scalar() (in module ufl.tensors), 162
as_scalars() (in module ufl.tensors), 162
as_tensor() (in module ufl), 177
as_tensor() (in module ufl.tensors), 162
as_ufl() (in module ufl), 177
as_ufl() (in module ufl.constantvalue), 134
as_vector() (in module ufl), 177
as_vector() (in module ufl.tensors), 162
Asin (class in ufl.classes), 119
Asin (class in ufl.mathfunctions), 150
asin() (in module ufl), 178
asin() (in module ufl.operands), 153
asin() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
asin() (ufl.algorithms.forward_ad.ForwardAD method), 67
asin() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
atan() (in module ufl), 179
atan() (in module ufl.operands), 153
atan() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
atan() (ufl.algorithms.forward_ad.ForwardAD method), 67
atan() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
atan() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
atan2() (in module ufl), 179
atan2() (in module ufl.operands), 153
atan2() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
atan2() (ufl.algorithms.forward_ad.ForwardAD method), 67
atan2() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
atan2() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
atan_2() (in module ufl), 179
atan_2() (in module ufl.operands), 153
atan_2() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
atan_2() (ufl.algorithms.forward_ad.ForwardAD method), 67
atan_2() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
attach_implementations_of_indexing_interface() (in module ufl.core.ufldtype), 79
attach_operators_from_hash_data() (in module ufl.core.ufldtype), 79
attach_ufl_id() (in module ufl.core.ufldtype), 79
avg() (in module ufl), 181
avg() (in module ufl.operands), 153
B
begin() (ufl.log.Logger method), 149

Index
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bessel_function()</code></td>
<td><code>ufl.algorithms.estimate_degrees.SumDegreeEstimator</code></td>
<td>61</td>
</tr>
<tr>
<td><code>bessel_function()</code></td>
<td><code>ufl.algorithms.pdiffs.PartialDerivativeComputer</code></td>
<td>71</td>
</tr>
<tr>
<td><code>bessel_I()</code></td>
<td><code>ufl.algorithms.apply_derivatives.GenericDerivativeRuleset</code></td>
<td>49</td>
</tr>
<tr>
<td><code>bessel_i()</code></td>
<td><code>ufl.algorithms.forward_ad.ForwardAD</code></td>
<td>67</td>
</tr>
<tr>
<td><code>bessel_i()</code></td>
<td><code>ufl.formatting.ufl2latex.Expression2LatexHandler</code></td>
<td>91</td>
</tr>
<tr>
<td><code>bessel_J()</code></td>
<td><code>ufl.algorithms.apply_derivatives.GenericDerivativeRuleset</code></td>
<td>49</td>
</tr>
<tr>
<td><code>bessel_j()</code></td>
<td><code>ufl.algorithms.forward_ad.ForwardAD</code></td>
<td>67</td>
</tr>
<tr>
<td><code>bessel_j()</code></td>
<td><code>ufl.formatting.ufl2latex.Expression2LatexHandler</code></td>
<td>91</td>
</tr>
<tr>
<td><code>bessel_K()</code></td>
<td><code>ufl.algorithms.apply_derivatives.GenericDerivativeRuleset</code></td>
<td>49</td>
</tr>
<tr>
<td><code>bessel_k()</code></td>
<td><code>ufl.algorithms.forward_ad.ForwardAD</code></td>
<td>68</td>
</tr>
<tr>
<td><code>bessel_K()</code></td>
<td><code>ufl.formatting.ufl2latex.Expression2LatexHandler</code></td>
<td>91</td>
</tr>
<tr>
<td><code>bessel_Y()</code></td>
<td><code>ufl.algorithms.apply_derivatives.GenericDerivativeRuleset</code></td>
<td>49</td>
</tr>
<tr>
<td><code>bessel_y()</code></td>
<td><code>ufl.algorithms.forward_ad.ForwardAD</code></td>
<td>68</td>
</tr>
<tr>
<td><code>bessel_y()</code></td>
<td><code>ufl.formatting.ufl2latex.Expression2LatexHandler</code></td>
<td>91</td>
</tr>
</tbody>
</table>

**Index**

191
cell_avg() (ufl.algorithms.check_arities.ArityChecker method), 58
cell_avg() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 61
cell_avg() (ufl.algorithms.formtransformations.PartExtracter method), 65
(cell_avg() (ufl.algorithms.forward_ad.ForwardAD method), 68
(cell_avg() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
cell_avg() (ufl.formatting.ufl2dot.CompactLabeller method), 89
cell_avg() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
cell_coordinate() (ufl.algorithms.apply_derivatives.GradRuleset method), 50
cell_coordinate() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 51
cell_coordinate() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 52
cell_coordinate() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 53
cell_coordinate() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 61
cell_coordinate() (ufl.algorithms.forward_ad.GradAD method), 69
cell_edge_vectors() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_facet_jacobian() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
(cell_facet_jacobian_determinant() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
(cell_facet_jacobian_inverse() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_facet_origin() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_normal() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_orientation() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_origin() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
cell_volume() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 52
cell_volume() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
CellAvg (class in ufl.classes), 114
CellAvg (class in ufl.restriction), 158
CellCoordinate (class in ufl.classes), 103
CellCoordinate (class in ufl.geometry), 142
CellEdgeVectors (class in ufl.classes), 105
CellEdgeVectors (class in ufl.geometry), 142
CellFacetJacobian (class in ufl.classes), 105
CellFacetJacobian (class in ufl.geometry), 142
CellFacetJacobianDeterminant (class in ufl.classes), 106
CellFacetJacobianDeterminant (class in ufl.geometry), 142
CellFacetJacobianInverse (class in ufl.classes), 106
CellFacetJacobianInverse (class in ufl.geometry), 142
CellFacetOrigin (class in ufl.classes), 104
CellFacetOrigin (class in ufl.geometry), 143
cellname() (ufl.Cell method), 99
cellname() (ufl.classes.Cell method), 120
CellNormal (class in ufl), 169
CellNormal (class in ufl.classes), 106
CellNormal (class in ufl.geometry), 143
CellOrientation (class in ufl.classes), 108
CellOrientation (class in ufl.geometry), 143
CellOrigin (class in ufl.classes), 104
CellOrientation (class in ufl.geometry), 143
CellVolume (class in ufl), 169
CellVolume (class in ufl.classes), 107
CellVolume (class in ufl.geometry), 143
cfname() (in module ufl.formatting.ufl2latex), 92
change_integrand_geometry_representation() (in module ufl.algorithms.change_to_reference), 58
change_regularity() (in module ufl.algorithms.elementtransformations), 61
change_to_reference_grad() (in module ufl.algorithms.change_to_reference), 58
check_abstract_trait_consistency() (in module ufl.core.ufl_type), 79
check_form_arity() (in module ufl.algorithms.check_arities), 59
check_has_slots() (in module ufl.core.ufl_type), 79
check_implements_required_properties() (in module ufl.core.ufl_type), 79
circumradius() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 52
Circumradius (class in ufl), 169
Circumradius (class in ufl.classes), 107
Circumradius (class in ufl.geometry), 143
Circumradius() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 52
CompoundTensorOperator (class in ufl.tensoralgebra), 160
compute_coefficient_forward_ad() (in module ufl.algorithms.forward_ad), 70
compute_derivative_tuples() (in module ufl.utils.derivativetuples), 94
compute_energy_norm() (in module ufl.algorithms.formtransformations), 66
compute_expr_hash() (in module ufl.core.compute_expr_hash), 75
compute_expression_hashdata() (in module ufl.algorithms.signature), 73
compute_expression_signature() (in module ufl.algorithms.signature), 73
compute_form_action() (in module ufl.algorithms.formtransformations), 66
compute_form_adjoint() (in module ufl.algorithms.formtransformations), 66
compute_form_arities() (in module ufl.algorithms.formtransformations), 67
compute_form_data() (in module ufl.algorithms.compute_form_data), 59
compute_form_functional() (in module ufl.algorithms.formtransformations), 67
compute_form_lhs() (in module ufl.algorithms.formtransformations), 67
compute_form_rhs() (in module ufl.algorithms.formtransformations), 67
compute_form_signature() (in module ufl.algorithms.signature), 73
compute_form_with_arity() (in module ufl.algorithms.formtransformations), 67
compute_grad_forward_ad() (in module ufl.algorithms.forward_ad), 70
compute_indices() (in module ufl.permutation), 157
compute_indices2() (in module ufl.permutation), 157
compute_integrand_scaling_factor() (in module ufl.algorithms.apply_integral_scaling), 53
compute_multiindex_hashdata() (in module ufl.algorithms.signature), 73
compute_order_tuples() (in module ufl.permutation), 157
compute_permutation_pairs() (in module ufl.permutation), 157
compute_permutations() (in module ufl.permutation), 157
compute_sign() (in module ufl.permutation), 157
compute_terminal_hashdata() (in module ufl.algorithms.signature), 73
compute_variable_forward_ad() (in module ufl.algorithms.forward_ad), 70
Condition (class in ufl.classes), 117
Condition (class in ufl.conditional), 131
condition() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
Conditional (class in ufl.classes), 118
Conditional (class in ufl.conditional), 132
conditional() (in module ufl), 179
conditional() (in module ufl.operators), 154
conditional() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
constant() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 61
constant() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
Constant() (in module ufl), 176
Constant() (in module ufl.coefficient), 130
Constant() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 61
ConstantValue (class in ufl.classes), 109
ConstantValue (class in ufl.constantvalue), 133
cos() (in module ufl), 178
cos() (in module ufl.operators), 154
cos() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
cos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
coordinates() (in module ufl.classes.Mesh), 126
coordinates() (in module ufl.domain.Mesh), 136
drift() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
CopyTransformer (class in ufl.algorithms.transformer), 73
Cosh (class in ufl.classes), 118
Cosh (class in ufl.mathfunctions), 151
cosh() (ufl.algorithms.forward_ad.ForwardAD method), 68
cosh() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
cosh() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
CopyTransformer (class in ufl.algorithms.transformer), 73
Cos (class in ufl.classes), 118
Cosh (class in ufl.classes), 119
Cosh (class in ufl.mathfunctions), 151
cosh() (in module ufl), 179
cosh() (in module ufl.operators), 154
cosh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
coordinates() (ufl.Mesh method), 168
coordinates() (ufl.Mesh method), 168
Cosh (class in ufl.classes), 119
Cos (class in ufl.classes), 118
Cosh (class in ufl.mathfunctions), 151
cosh() (in module ufl), 179
cosh() (in module ufl.operators), 154
cosh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
cosh() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
cosh() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
count() (ufl.classes.Coefficient method), 110
count() (ufl.classes.Index method), 127
count() (ufl.classes.Label method), 111
count() (ufl.Coefficient method), 176
count() (ufl.coefficient.Coefficient method), 130
count() (ufl.core.multindex.Index method), 77
count() (ufl.Index method), 177
count() (ufl.utils.counted.ExampleCounted method), 93
counted_init() (in module ufl.utils.counted), 93
create_nested_lists() (in module ufl.algorithms.apply_function_pullbacks), 52
create_slice_indices() (in module ufl.index_combination_utils), 147
Cross (class in ufl.classes), 113
Cross (class in ufl.tensoralgebra), 160
cross() (in module ufl), 177
cross() (in module ufl.corealg), 154
cross() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cross() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 56
cross() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
cross() (ufl.algorithms.forward_ad.UnimplementedADRules method), 69
cross() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
cross_expr() (in module ufl.compound_expressions), 131
Curl (class in ufl.classes), 116
Curl (class in ufl.differentiation), 134
curl() (in module ufl), 180
curl() (in module ufl.corealg), 154
curl() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
curl() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 56
curl() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
curl() (ufl.algorithms.forward_ad.UnimplementedADRules method), 70
curl() (ufl.formatting.ufl2dot.CompressLabeller method), 89
curl() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
cutoff_post_traversal() (in module ufl.corealg.traversal), 81

cosh() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 71
cosh() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
count() (ufl.classes.Coefficient method), 110
count() (ufl.classes.Index method), 127
count() (ufl.classes.Label method), 111
count() (ufl.Coefficient method), 176
count() (ufl.coefficient.Coefficient method), 130
count() (ufl.core.multindex.Index method), 77
count() (ufl.Index method), 177
count() (ufl.utils.counted.ExampleCounted method), 93
counted_init() (in module ufl.utils.counted), 93
count() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cross() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 56
cross() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
cross() (ufl.algorithms.forward_ad.UnimplementedADRules method), 69
cross() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
cross_expr() (in module ufl.compound_expressions), 131
cross_expr_2x2() (in module ufl.compound_expressions), 131
<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr()</td>
<td>ufl.algorithms.check_restrictions.RestrictionChecker</td>
<td>59</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.estimate_degrees.SumDegreeEstimator</td>
<td>62</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.formtransformations.PartExtractor</td>
<td>65</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.forward_ad.ForwardAD</td>
<td>68</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.pdiffs.PartialDerivativeComputer</td>
<td>71</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.replace.Replacer</td>
<td>73</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.transformer.CopyTransformer</td>
<td>73</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.transformer.ReuseTransformer</td>
<td>74</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.algorithms.transformer.Transformer</td>
<td>74</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.corealg.multifunction.MultiFunction</td>
<td>80</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.formatting.graph.StringDependencyDefiner</td>
<td>88</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.formatting.ufl2latex.Expression2LatexHandler</td>
<td>90</td>
</tr>
<tr>
<td>expr()</td>
<td>ufl.formatting.ufl2latex.ExprData</td>
<td>138</td>
</tr>
<tr>
<td>expression()</td>
<td>ufl.classes.Variable</td>
<td>111</td>
</tr>
<tr>
<td>expression()</td>
<td>ufl.classes.Variable</td>
<td>111</td>
</tr>
<tr>
<td>expression2latex()</td>
<td>ufl.classes.Variable</td>
<td>163</td>
</tr>
<tr>
<td>extract_component()</td>
<td>ufl.classes.FiniteElementBase</td>
<td>121</td>
</tr>
<tr>
<td>extract_component()</td>
<td>ufl.classes.MixedElement</td>
<td>122</td>
</tr>
<tr>
<td>extract_component()</td>
<td>ufl.finiteelement.finiteelementbase.FiniteElementBase</td>
<td>83</td>
</tr>
<tr>
<td>extract_component()</td>
<td>ufl.finiteelement.mixedelement.MixedElement</td>
<td>84</td>
</tr>
<tr>
<td>extract_component()</td>
<td>ufl.FiniteElementBase</td>
<td>170</td>
</tr>
<tr>
<td>extract_domains()</td>
<td>ufl.domain</td>
<td>137</td>
</tr>
<tr>
<td>extract_elements()</td>
<td>ufl.algorithms.analysis</td>
<td>46</td>
</tr>
<tr>
<td>extract_incoming_edges()</td>
<td>ufl.formatting.graph</td>
<td>88</td>
</tr>
<tr>
<td>extract_incoming_vertex_connections()</td>
<td>ufl.formatting.graph</td>
<td>88</td>
</tr>
<tr>
<td>extract_outgoing_edges()</td>
<td>ufl.formatting.graph</td>
<td>88</td>
</tr>
<tr>
<td>extract_outgoing_vertex_connections()</td>
<td>ufl.formatting.graph</td>
<td>88</td>
</tr>
<tr>
<td>extract_reference_component()</td>
<td>ufl.classes.FiniteElementBase</td>
<td>121</td>
</tr>
<tr>
<td>extract_reference_component()</td>
<td>ufl.classes.MixedElement</td>
<td>122</td>
</tr>
<tr>
<td>extract_sub_elements()</td>
<td>ufl.algorithms.analysis</td>
<td>46</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.classes.FiniteElementBase</td>
<td>121</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.classes.MixedElement</td>
<td>122</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.finiteelement.finiteelementbase.FiniteElementBase</td>
<td>83</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.finiteelement.mixedelement.MixedElement</td>
<td>84</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.finiteelement.mixedelement.TensorElement</td>
<td>85</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.FiniteElementBase</td>
<td>170</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.MixedElement</td>
<td>172</td>
</tr>
<tr>
<td>extract_subelement_component()</td>
<td>ufl.TensorElement</td>
<td>172</td>
</tr>
</tbody>
</table>
Unified Form Language (UFL) Documentation, Release 1.7.0dev

Index

FacetCoordinate (class in ufl.classes), 104
FacetCoordinate (class in ufl.geometry), 143
FacetEdgeVectors (class in ufl.classes), 105
FacetEdgeVectors (class in ufl.geometry), 144
FacetElement (class in ufl), 174
FacetElement (class in ufl.classes), 125
FacetElement (class in ufl.finiteelement.facetelement), 82
FacetJacobian (class in ufl.classes), 104
FacetJacobian (class in ufl.geometry), 144
FacetJacobianDeterminant (class in ufl.classes), 105
FacetJacobianDeterminant (class in ufl.geometry), 144
FacetJacobianInverse (class in ufl.classes), 106
FacetJacobianInverse (class in ufl.geometry), 144
FacetNormal (class in ufl.classes), 104
FacetNormal (class in ufl.geometry), 144
FacetOrientation (class in ufl.classes), 108
FacetOrientation (class in ufl.geometry), 145
FacetOrigin (class in ufl.classes), 104
FacetOrigin (class in ufl.geometry), 145
family() (ufl.classes.FiniteElementBase method), 121
family() (ufl.finiteelement.mixedelement.TensorElement method), 83
family() (uflFINITEELEMENTFINITEELEMENTBASE.FiniteElementBase method), 83
FileData (class in ufl.formdata), 65
find_geometric_dimension() (in module ufl.domain), 137
FiniteElement (class in ufl), 171
FiniteElement (class in ufl.classes), 122
FiniteElement (class in ufl.finiteelement.finiteelement), 82
FiniteElementBase (class in ufl), 170
FiniteElementBase (class in ufl.classes), 121
FiniteElementBase (class in ufl.finiteelement.finiteelementbase), 83
FixedIndex (class in ufl.classes), 127
FixedIndex (class in ufl.core.multiindex), 77
fixme() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
flatten_multiindex() (in module ufl.utilz.indexflattening), 95
flattened_sub_element_mapping() (ufl.classes.TensorElement method), 123
flattened_sub_element_mapping() (ufl.finiteelement.mixedelement.TensorElement method), 85
flattened_sub_element_mapping() (ufl.TensorElement method), 172
FloatValue (class in ufl.classes), 109
FloatValue (class in ufl.constanvalue), 133
Form (class in ufl), 181
Form (class in ufl.classes), 129
Form (class in ufl.form), 138
form2code2latex() (in module ufl.formatters.ufl2latex), 93
form2latex() (in module ufl.formatters.ufl2latex), 93
form_argument() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
form_argument() (ufl.algorithms.apply_function_pullbacks.FunctionPullbackApplier method), 52
form_argument() (ufl.algorithms.change_to_reference.NEWChangeToReference method), 57
form_argument() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
form_argument() (ufl.algorithms.expand_indices.IndexExpander method), 64
form_argument() (ufl.formatters.ufl2latex.Compactlabeller method), 89
form_info() (in module ufl.formatters.printing), 89
FormArgument (class in ufl.classes), 103
Format (class in ufl.core.terminal), 78
format_entities() (in module ufl.formatting.ufl2dot), 90
format_float() (in module ufl.constantvalue), 134
format_index() (in module ufl.formatters.ufl2latex), 93
format_multi_index() (in module ufl.formatters.ufl2latex), 93
FormData (class in ufl.formdata), 64
formdata2latex() (in module ufl.formatters.ufl2latex), 93
forms2latexdocument() (in module ufl.formatters.ufl2latex), 93
ForwardAD (class in ufl.algorithms.forward_ad), 67
free_indices() (ufl.classes.Expr method), 102
free_indices() (ufl.classes.ExprList method), 115
free_indices() (ufl.classes.ExprMapping method), 115
free_indices() (ufl.core.expr.Expr method), 76
free_indices() (ufl.exprrcontainers.ExprList method), 137
free_indices() (ufl.exprrcontainers.ExprMapping method), 137
from_numpy_to_lists() (in module ufl.tensors), 162
functional() (in module ufl), 184
functional() (in module ufl.forward_ad), 140
functionSpace (class in ufl.functionspace), 141
GateauxDerivativeRuleset (class in ufl.algorithms.apply_derivatives), 48
GE (class in ufl.classes), 117
GE (class in ufl.conditionals), 132
g() (in module ufl), 179
g() (in module ufl.conditionals), 155

G

Index
ge() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 91
generic_pseudo_determinant_expr() (in module ufl.compound_expressions), 131
generic_pseudo_inverse_expr() (in module ufl.compound_expressions), 131
GenericDerivativeRuleset (class in ufl.algorithms.apply_derivatives), 49
geometric_cell_quantity() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
geometric_dimension() (ufl.AbstractCell method), 166
geometric_dimension() (ufl.AbstractDomain method), 167
geometric Dimension() (ufl.cell.AbstractCell method), 99
geometric_dimension() (ufl.classes.AbstractCell method), 120
geometric_dimension() (ufl.classes.AbstractDomain method), 125
geometric_dimension() (ufl.classes.Expr method), 102
geometric_dimension() (ufl.classes.Form method), 129
geometric_dimension() (ufl.core.expr.Expr method), 77
geometric_dimension() (ufl.domain.AbstractDomain method), 136
geometric_dimension() (ufl.Form method), 182
geometric_dimension() (ufl.form.Form method), 139
geometric_facet_quantity() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
geometric_quantity() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 48
geometric_quantity() (ufl.algorithms.apply_derivatives.GenericsDerivativeRuleset method), 49
geometric_quantity() (ufl.algorithms.apply_derivatives.GradRuleset method), 50
geometric_quantity() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 51
geometric_quantity() (ufl.algorithms.apply_derivatives.VariableRuleset method), 51
geometric_quantity() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
geometric_quantity() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 58
geometric_quantity() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
geometric_quantity() (ufl.algorithms.formtransformations.PartialExtractor method), 64
geometric_quantity() (ufl.algorithms.forward_ad.GradAD method), 69
geometric_quantity() (ufl.algorithms.forward_ad.UnusedADRules method), 70
GeometricCellQuantity (class in ufl.classes), 103
GeometricCellQuantity (class in ufl.geometry), 145
GeometricFacetQuantity (class in ufl.classes), 103
GeometricFacetQuantity (class in ufl.geometry), 145
GeometricQuantity (class in ufl.classes), 103
GeometricQuantity (class in ufl.geometry), 145
GeometricLoweringApplier (class in ufl.algorithms.apply_geometry_lowering), 52
get_base_attr() (in module ufl.core.ufl_type), 79
get_handler() (ufl.log.Logger method), 149
get_logger() (ufl.log.Logger method), 149
get_num_args() (in module ufl.corealg.multifunction), 80
get_status_output() (in module ufl.utils.system), 96
Grad (class in ufl.classes), 116
Grad (class in ufl.differentiation), 135
grad() (in module ufl), 180
grad() (in module ufl.operators), 155
grad() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 48
grad() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 48
grad() (ufl.algorithms.apply_derivatives.GenericsDerivativeRuleset method), 49
grad() (ufl.algorithms.apply_derivatives.GradRuleset method), 50
grad() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 51
grad() (ufl.algorithms.apply_derivatives.VariableRuleset method), 51
grad() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 54
grad() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 66
grad() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
grad() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 58
grad() (ufl.algorithms.check_arities.ArityChecker method), 58
grad() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
grad() (ufl.algorithms.expand_indices.IndexExpander method), 64
grad() (ufl.algorithms.formtransformations.PartialExtractor method), 64
grad() (ufl.algorithms.forward_ad.CoefficientAD method), 67
grad() (ufl.algorithms.forward_ad.ForwardAD method), 68
grad() (ufl.algorithms.forward_ad.GradAD method), 69
grad() (ufl.algorithms.forward_ad.UnusedADRules method), 70
grad() (ufl.algorithms.forward_ad.VariableAD method), 70
grad() (ufl.formatting.ufl2dot.CompactLabeller method), 89
202 Index
grad() (ufl.formattting.ufl2latex.Expression2LatexHandler method), 91
GradAD (class in ufl.algorithms.forward_ad), 69
GradRuleset (class in ufl.algorithms.apply_derivatives), 50
Graph (class in ufl.formatting.graph), 87
group_integrals_by_domain_and_type() (in module ufl.algorithms.domain_analysis), 60
GT (class in ufl.classes), 117
GT (class in ufl.conditional), 132
gt() (in module ufl), 179
gt() (in module ufl.types), 155
gt() (ufl.formattting.ufl2latex.Expression2LatexHandler method), 91

H
has_exact_type() (in module ufl.algorithms.analysis), 47
has_simplex_facets() (ufl.AbstractCell method), 166
has_simplex_facets() (ufl.Cell method), 166
has_simplex_facets() (ufl.cell.AbstractCell method), 99
has_simplex_facets() (ufl.cell.Cell method), 99
has_simplex_facets() (ufl.cell.OuterProductCell method), 100
has_simplex_facets() (ufl.cell.TensorProductCell method), 100
has_simplex_facets() (ufl.classes.AbstractCell method), 120
has_simplex_facets() (ufl.classes.Cell method), 120
has_simplex_facets() (ufl.classes.OuterProductCell method), 121
has_simplex_facets() (ufl.classes.TensorProductCell method), 120
has_simplex_facets() (ufl.operators.Cell method), 167
has_simplex_facets() (ufl.operators.TensorProductCell method), 167
has_type() (in module ufl.algorithms.analysis), 47
HcurlElement (class in ufl), 174
HcurlElement (class in ufl.classes), 125
HcurlElement (class in ufl.finiteelement.hdivcurl), 84
HdivElement (class in ufl), 174
HdivElement (class in ufl.classes), 125
HdivElement (class in ufl.finiteelement.hdivcurl), 84
HeapItem (class in ufl.formatting.graph), 88
hypercube() (in module ufl.cell), 100

I
id_or_none() (in module ufl.protocols), 158
Identity (class in ufl), 176
Identity (class in ufl.classes), 109
Identity (class in ufl.constantvalue), 133
identity() (ufl.formattting.ufl2dot.CompactLabeller method), 89
identity() (ufl.formattting.ufl2latex.Expression2LatexHandler method), 91
increase_order() (in module ufl.algorithms.elementtransformations), 61
independent_operator() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
independent_terminal() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
Index (class in ufl), 177
Index (class in ufl.classes), 127
Index (class in ufl.core.multiindex), 77
index() (ufl.algorithms.renumbering.IndexRenumberingTransformer method), 72
index() (ufl.classes.IndexSum method), 114
index() (ufl.indexsum.IndexSum method), 148
index_dimensions() (ufl.classes.Expr method), 115
index_dimensions() (ufl.classes.ExprList method), 102
index_dimensions() (ufl.classes.ExprMapping method), 115
index_dimensions() (ufl.core.expr.Expr method), 77
index_dimensions() (ufl.expcontaniers.ExprList method), 137
index_dimensions() (ufl.expcontaniers.ExprMapping method), 137
index_sum() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
index_sum() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 56
index_sum() (ufl.algorithms.check_arities.ArityChecker method), 58
index_sum() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
index_sum() (ufl.algorithms.expand_indices.IndexExpander method), 64
index_sum() (ufl.algorithms.formtransformations.PartExtractor method), 65
index_sum() (ufl.algorithms.forward_ad.ForwardAD method), 68
index_sum() (ufl.algorithms.pdiffs.PartialDerivativeComputer method), 72
index_sum() (ufl.formattting.ufl2dot.CompactLabeller method), 89
index_sum() (ufl.formattting.ufl2latex.Expression2LatexHandler method), 91
IndexBase (class in ufl.classes), 127
IndexBase (class in ufl.core.multiindex), 77
Indexed (class in ufl.classes), 109
Indexed (class in ufl.indexed), 147
indexed() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 48
indexed() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
indexed() (ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 56
indexed() (ufl.algorithms.check_arities.ArityChecker method), 58
is_cellwise_constant() (ufl.classes.Argument method), 110
is_cellwise_constant() (ufl.classes.CellCoordinate method), 103
is_cellwise_constant() (ufl.classes.CellEdgeVectors method), 105
is_cellwise_constant() (ufl.classes.CellFacetJacobian method), 105
is_cellwise_constant() (ufl.classes.CellFacetJacobianDeterminant method), 106
is_cellwise_constant() (ufl.classes.CellFacetJacobianInverse method), 106
is_cellwise_constant() (ufl.classes.CellOrigin method), 104
is_cellwise_constant() (ufl.classes.Coefficient method), 111
is_cellwise_constant() (ufl.classes.ConstantValue method), 109
is_cellwise_constant() (ufl.classes.EnrichedElement method), 123
is_cellwise_constant() (ufl.classes.Expr method), 102
is_cellwise_constant() (ufl.classes.FacetCoordinate method), 104
is_cellwise_constant() (ufl.classes.FacetEdgeVectors method), 105
is_cellwise_constant() (ufl.classes.FacetJacobian method), 105
is_cellwise_constant() (ufl.classes.FacetJacobianDeterminant method), 106
is_cellwise_constant() (ufl.classes.FacetJacobianInverse method), 106
is_cellwise_constant() (ufl.classes.FacetNormal method), 106
is_cellwise_constant() (ufl.classes.FiniteElementBase method), 121
is_cellwise_constant() (ufl.classes.GeometricQuantity method), 103
is_cellwise_constant() (ufl.classes.Jacobian method), 104
is_cellwise_constant() (ufl.classes.JacobianDeterminant method), 105
is_cellwise_constant() (ufl.classes.Label method), 111
is_cellwise_constant() (ufl.classes.MixedElement method), 122
is_cellwise_constant() (ufl.classes.MultiIndex method), 108
is_cellwise_constant() (ufl.classes.QuadratureWeight method), 108
is_cellwise_constant() (ufl.classes.RestrictedElement method), 124
is_cellwise_constant() (ufl.classes.SpatialCoordinate method), 103
is_cellwise_constant() (ufl.coefficient.Coefficient method), 130
is_cellwise_constant() (ufl.constantvalue.ConstantValue method), 133
is_cellwise_constant() (ufl.core.expr.Expr method), 77
is_cellwise_constant() (ufl.core.multiindex.MultiIndex method), 77
is_cellwise_constant() (ufl.EnrichedElement method), 172
is_cellwise_constant() (ufl.FacetNormal method), 169
is_cellwise_constant() (ufl.FiniteElementBase method), 171
is_cellwise_constant() (ufl.geometry.CellCoordinate method), 142
is_cellwise_constant() (ufl.geometry.CellEdgeVectors method), 142
is_cellwise_constant() (ufl.geometry.CellFacetJacobian method), 142
is_cellwise_constant() (ufl.geometry.CellFacetJacobianDeterminant method), 142
is_cellwise_constant() (ufl.geometry.CellFacetJacobianInverse method), 142
is_cellwise_constant() (ufl.geometry.CellOrigin method), 143
is_cellwise_constant() (ufl.geometry.FacetCoordinate method), 144
is_cellwise_constant() (ufl.geometry.FacetEdgeVectors method), 144
is_cellwise_constant() (ufl.geometry.FacetJacobian method), 144
is_cellwise_constant() (ufl.geometry.FacetJacobianDeterminant method), 144
is_cellwise_constant() (ufl.geometry.FacetJacobianInverse method), 144
is_cellwise_constant() (ufl.geometry.FacetNormal method), 144
is_cellwise_constant() (ufl.geometry.GeometricQuantity method), 145
is_cellwise_constant() (ufl.geometry.Jacobian method), 145
is_cellwise_constant() (ufl.geometry.JacobianDeterminant method), 145
is_cellwise_constant() (ufl.geometry.JacobianInverse method), 146
is_cellwise_constant() (ufl.geometry.QuadratureWeight method), 146
is_cellwise_constant() (ufl.geometry.SpatialCoordinate method), 147
is_cellwise_constant() (ufl.Jacobian method), 170
is_cellwise_constant() (ufl.JacobianDeterminant method), 170
is_cellwise_constant() (ufl.JacobianInverse method), 170
is_cellwise_constant() (ufl.MixedElement method), 172
is_cellwise_constant() (ufl.RestrictedElement method), 173
is_cellwise_constant() (ufl.SpatialCoordinate method), 168
is_cellwise_constant() (ufl.variable.Label method), 163
is_globally_constant() (in module ufl.checks), 101
is_multilinear() (in module ufl.algorithms.predicates), 72
is_piecewise_linear_simplex_domain() (ufl.classes.Mesh method), 126
is_piecewise_linear_simplex_domain() (ufl.classes.MeshView method), 126
is_piecewise_linear_simplex_domain() (ufl.classes.TensorProductMesh method), 126
is_piecewise_linear_simplex_domain() (ufl.domain.Mesh method), 136
is_piecewise_linear_simplex_domain() (ufl.domain.MeshView method), 136
is_piecewise_linear_simplex_domain() (ufl.domain.TensorProductMesh method), 136
is_piecewise_linear_simplex_domain() (ufl.Mesh method), 168
is_piecewise_linear_simplex_domain() (ufl.MeshView method), 168
is_piecewise_linear_simplex_domain() (ufl.TensorProductMesh method), 168
is_post_handler() (in module ufl.algorithms.transformer), 75
is_python_scalar() (in module ufl.checks), 101
is_scalar_constant_expression() (in module ufl.checks), 101
is_simplex() (ufl.AbstractCell method), 166
is_simplex() (ufl.Cell method), 166
is_simplex() (ufl.cell.AbstractCell method), 99
is_simplex() (ufl.cell.Cell method), 99
is_simplex() (ufl.cell.OuterProductCell method), 100
is_simplex() (ufl.cell.TensorProductCell method), 100
is_simplex() (ufl.classes.AbstractCell method), 120
is_simplex() (ufl.classes.Cell method), 120
is_simplex() (ufl.classes.OuterProductCell method), 121
is_simplex() (ufl.classes.TensorProductCell method), 120
is_simplex() (ufl.OuterProductCell method), 167
is_simplex() (ufl.TensorProductCell method), 167
is_true_ufl_scalar() (in module ufl.checks), 101
is_ufl_scalar() (in module ufl.checks), 101
istr() (in module ufl.utils.formatting), 95

itemize() (in module ufl.formatting.latextools), 89
iter_expressions() (in module ufl.algorithms.traversal), 75
iter_tree() (in module ufl.utils.sequences), 95

J
Jacobian (class in ufl), 170
Jacobian (class in ufl.classes), 104
Jacobian (class in ufl.geometry), 145
jacobian() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
jacobian() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
jacobian() (ufl.algorithms.forward_ad.GradAD method), 69
jacobian_determinant() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
jacobian_determinant() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
jacobian_determinant() (ufl.algorithms.forward_ad.GradAD method), 69
jacobian_inverse() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
jacobian_inverse() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
jacobian_inverse() (ufl.algorithms.forward_ad.GradAD method), 69
JacobianDeterminant (class in ufl), 170
JacobianDeterminant (class in ufl.classes), 105
JacobianDeterminant (class in ufl.geometry), 145
JacobianInverse (class in ufl), 170
JacobianInverse (class in ufl.classes), 106
JacobianInverse (class in ufl.geometry), 146
join_domains() (in module ufl.domain), 137
jump() (in module ufl), 181
jump() (in module ufl.operators), 155

L
Label (class in ufl.classes), 111
Label (class in ufl.variable), 163
label() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
label() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
label() (ufl.classes.Variable method), 111
label() (ufl.variable.Variable method), 163
LE (class in ufl.classes), 117
LE (class in ufl.conditional), 132
le() (in module ufl), 179
le() (in module ufl.operators), 155
le() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 92
len_items() (in module ufl.formatting.graph), 88
lhs() (in module ufl), 184
lhs() (in module ufl.formoperators), 140
Index
Index
replace_include_statements() (in module ufl.algorithms.formfiles), 65
replace_integral_domains() (in module ufl), 184
replace_integral_domains() (in module ufl.form), 139
Replacer (class in ufl.algorithms.replace), 73
ReprLabeller (class in ufl.formattting.ufl2dot), 90
reshape_to_nested_list() (in module ufl.algorithms.apply_function_pullbacks), 52
Restricted (class in ufl.classes), 114
Restricted (class in ufl.restriction), 159
restricted() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
restricted() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
restricted() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
restricted() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
restricted() (ufl.algorithms.forward_ad.ForwardAD method), 68
restricted_sub_elements() (ufl.classes.RestrictedElement method), 124
restricted_sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method), 86
restricted_sub_elements() (ufl.RestrictedElement method), 173
RestrictedElement (class in ufl), 173
RestrictedElement (class in ufl.classes), 123
RestrictedElement (class in ufl.finiteelement.restrictedelement), 86
restriction_domain() (ufl.classes.RestrictedElement method), 124
restriction_domain() (ufl.finiteelement.restrictedelement.RestrictedElement method), 86
restriction_domain() (ufl.RestrictedElement method), 173
RestrictionChecker (class in ufl.algorithms.check_restrictions), 59
RestrictionPropagator (class in ufl.algorithms.apply_restrictions), 53
reuse() (ufl.algorithms.transformer.Transformer method), 74
reuse_if_possible() (ufl.algorithms.transformer.Transformer method), 74
reuse_if_untouched() (ufl.algorithms.transformer.Transformer method), 74
reuse_variable() (ufl.algorithms.transformer.Transformer method), 74
ReuseTransformer (class in ufl.algorithms.transformer), 74
rhs() (in module ufl), 184
rhs() (in module ufl.formoperators), 140
rot() (in module ufl), 180
rot() (in module ufl.operatrs), 156
ScalarValue (class in ufl.classes), 109
ScalarValue (class in ufl.constantvalue), 133
sensitivity_rhs() (in module ufl.formoperators), 140
set_handler() (ufl.log.Logger method), 149
set_level() (ufl.log.Logger method), 150
set_list_item() (in module ufl.formoperators), 141
set_prefix() (ufl.log.Logger method), 150
set_trait() (in module ufl.core.ufl_type), 79
shape() (in module ufl), 177
shape() (in module ufl.operatrs), 156
shape() (ufl.classes.Expr method), 102
shape() (ufl.core.expr.Expression method), 77
shape_to_strides() (in module ufl.utils.indexflattening), 95
shortstr() (ufl.BrokenElement method), 174
shortstr() (ufl.classes.BrokenElement method), 125
shortstr() (ufl.classes.EnrichedElement method), 123
shortstr() (ufl.classes.FacetElement method), 125
shortstr() (ufl.classes.FiniteElement method), 122
shortstr() (ufl.classes.HCurlElement method), 125
shortstr() (ufl.classes.HDivElement method), 125
shortstr() (ufl.classes.InteriorElement method), 125
shortstr() (ufl.classes.MixedElement method), 123
shortstr() (ufl.classes.OuterProductElement method), 124
shortstr() (ufl.classes.OuterProductVectorElement method), 125
shortstr() (ufl.classes.RestrictedElement method), 124
shortstr() (ufl.classes.TensorElement method), 123
shortstr() (ufl.classes.TensorProductElement method), 124
shortstr() (ufl.classes.TraceElement method), 125
shortstr() (ufl.classes.VectorElement method), 123
shortstr() (ufl.EnrichedElement method), 173
shortstr() (ufl.FacetElement method), 174
shortstr() (ufl.FiniteElement method), 171
shortstr() (ufl.finiteelement.brokenelement.BrokenElement method), 81
shortstr() (ufl.finiteelement.enrichedelement.EnrichedElement method), 82
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum()</td>
<td>ufl.formatting.ufl2dot.CompactLabeller method, 90</td>
</tr>
<tr>
<td>sum()</td>
<td>ufl.formatting.ufl2latex.Expression2LatexHandler method, 92</td>
</tr>
<tr>
<td>SumDegreeEstimator</td>
<td>class in ufl.algorithms.estimate_degrees, 61</td>
</tr>
<tr>
<td>Sym</td>
<td>class in ufl.classes, 114</td>
</tr>
<tr>
<td>SumDegreeEstimator</td>
<td>class in ufl.tensoralgebra, 161</td>
</tr>
<tr>
<td>sym()</td>
<td>(in module ufl), 178</td>
</tr>
<tr>
<td>sym()</td>
<td>(in module ufl.operators), 157</td>
</tr>
<tr>
<td>sym()</td>
<td>(ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 48</td>
</tr>
<tr>
<td>sym()</td>
<td>(ufl.algorithms.argument_dependencies.ArgumentDependencyExtracter method), 57</td>
</tr>
<tr>
<td>sym()</td>
<td>(ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63</td>
</tr>
<tr>
<td>sym()</td>
<td>(ufl.formatting.ufl2latex.Expression2LatexHandler method), 92</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.classes.FiniteElementBase method), 122</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.classes.MixedElement method), 123</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.classes.RestrictedElement method), 124</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.classes.TensorElement method), 123</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.finiteelement.finiteelementbase.FiniteElementElementBase method), 83</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.finiteelement.mixedelement.MixedElement method), 85</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.finiteelement.mixedelement.TensorElement method), 85</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.finiteelement.restrictedelement.RestrictedElement method), 86</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.FiniteElementBase method), 171</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.MixedElement method), 172</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.RestrictedElement method), 173</td>
</tr>
<tr>
<td>symmetry()</td>
<td>(ufl.TensorElement method), 172</td>
</tr>
<tr>
<td>system()</td>
<td>(in module ufl), 184</td>
</tr>
<tr>
<td>system()</td>
<td>(in module ufl.formoperators), 141</td>
</tr>
<tr>
<td>T (ufl.classes.Expr attribute), 102</td>
<td></td>
</tr>
<tr>
<td>T (ufl.core.expr.Expr attribute), 76</td>
<td></td>
</tr>
<tr>
<td>Tan</td>
<td>(class in ufl.classes), 119</td>
</tr>
<tr>
<td>Tan</td>
<td>(class in ufl.mathfunctions), 151</td>
</tr>
<tr>
<td>tan()</td>
<td>(in module ufl), 178</td>
</tr>
<tr>
<td>tan()</td>
<td>(in module ufl.operators), 157</td>
</tr>
<tr>
<td>tan()</td>
<td>(ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50</td>
</tr>
<tr>
<td>tanh()</td>
<td>(module ufl.operators), 157</td>
</tr>
<tr>
<td>tanh()</td>
<td>(ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50</td>
</tr>
<tr>
<td>tanh()</td>
<td>(ufl.algorithms.forward_ad.ForwardAD method), 68</td>
</tr>
<tr>
<td>tanh()</td>
<td>(ufl.algorithms.pdiffs.PartialDerivativeComputer method), 72</td>
</tr>
<tr>
<td>tanh()</td>
<td>(ufl.formatting.ufl2latex.Expression2LatexHandler method), 92</td>
</tr>
<tr>
<td>tanh()</td>
<td>(module ufl.algorithms.elementtransformations),</td>
</tr>
<tr>
<td>TensorConstant()</td>
<td>(in module ufl), 176</td>
</tr>
<tr>
<td>TensorConstant()</td>
<td>(in module ufl.coefficient), 130</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>(class in ufl), 172</td>
</tr>
<tr>
<td>TensorProductCell</td>
<td>(class in ufl), 167</td>
</tr>
<tr>
<td>TensorProductCell</td>
<td>(class in ufl.cell), 100</td>
</tr>
<tr>
<td>TensorProductCell</td>
<td>(class in ufl.classes), 120</td>
</tr>
<tr>
<td>TensorProductElement</td>
<td>(class in ufl), 173</td>
</tr>
<tr>
<td>TensorProductElement</td>
<td>(class in ufl.classes), 124</td>
</tr>
<tr>
<td>TensorProductElement</td>
<td>(class in ufl.finiteelement.tensorproductelement), 87</td>
</tr>
<tr>
<td>TensorProductFunctionSpace</td>
<td>(class in ufl.classes), 127</td>
</tr>
<tr>
<td>TensorProductFunctionSpace</td>
<td>(class in ufl.functionspace), 141</td>
</tr>
<tr>
<td>TensorProductMesh</td>
<td>(class in ufl), 168</td>
</tr>
<tr>
<td>TensorProductMesh</td>
<td>(class in ufl.classes), 126</td>
</tr>
<tr>
<td>TensorProductMesh</td>
<td>(class in ufl.domain), 136</td>
</tr>
<tr>
<td>Terminal</td>
<td>(class in ufl.classes), 102</td>
</tr>
<tr>
<td>Terminal</td>
<td>(class in ufl.core.terminal), 78</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.expand_indices.IndexExpander method), 52</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.argument_dependencies.ArgumentDependencyExtractor method), 57</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.change_to_reference.NEWChangeToReference method), 57</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.change_to_reference.OLDChangeToReference method), 58</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.check_arities.ArityChecker method), 59</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.expand_indices.IndexExpander method), 64</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.formtransformations.PartExtractor method), 66</td>
</tr>
<tr>
<td>terminal()</td>
<td>(ufl.algorithms.forward_ad.ForwardAD method), 68</td>
</tr>
</tbody>
</table>

Index 217
terminal() (ufl.algorithms.replace.Replacer method), 73
terminal() (ufl.algorithms.transformer.CopyTransformer method), 74
terminal() (ufl.algorithms.transformer.ReuseTransformer method), 74
terminal() (ufl.algorithms.transformer.Transformer method), 74
terminal() (ufl.formatting.ufl2dot.ReprLabeller method), 90
testdocument() (in module ufl.formatting.latextools), 89
TestFunction() (in module ufl), 175
TestFunction() (in module ufl.classes), 127
TestFunctions() (in module ufl), 175
TestFunctions() (in module ufl.argument), 98
tex2pdf() (in module ufl.formatting.ufl2latex), 93
Timer (class in ufl.utils.timer), 97
topological_dimension() (ufl.AbstractCell method), 166
topological_dimension() (ufl.AbstractDomain method), 167
topological_dimension() (ufl.cell.AbstractCell method), 99
topological_dimension() (ufl.classes.AbstractCell method), 120
topological_dimension() (ufl.classes.AbstractDomain method), 126
topological_dimension() (ufl.domain.AbstractDomain method), 136
topological_sorting() (in module ufl.utils.sorting), 96
tr() (in module ufl), 178
tr() (in module ufl.operators), 157
Trace (class in ufl.classes), 113
Trace (class in ufl.tensoralgebra), 161
trace() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 48
trace() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 48
trace() (ufl.algorithms.argument_dependencies.ArgumentDependency method), 57
transposed() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 48
transposed() (ufl.algorithms.argument_dependencies.ArgumentDependency method), 57
transposed() (ufl.algorithms.estimates_degrees.SumDegreeEstimator method), 63
transposed() (ufl.algorithms.forward_ad.UnusedADRules method), 70
transposed() (ufl.formatting.ufl2dot.CompactLabeller method), 90
transposed() (ufl.formatting.ufl2latex.Expression2LatexHandler method), 92
traverse_terminals() (in module ufl.corealg.traversal), 81
traverse_unique_terminals() (in module ufl.corealg.traversal), 81
tree_format() (in module ufl.formatting.printing), 89
TrialFunction() (in module ufl), 175
TrialFunction() (in module ufl.argument), 99
TrialFunction() (in module ufl.classes), 127
TrialFunctions() (in module ufl), 175
TrialFunctions() (in module ufl.argument), 99
TrialFunctions() (in module ufl.classes), 127
tstr() (in module ufl.utils.formatting), 95

U
ufl (module), 163
ufl.algebra (module), 97
ufl.algorithms (module), 75
ufl.algorithms.ad (module), 46
ufl.algorithms.analysis (module), 46
ufl.algorithms.apply_algebra_lowering (module), 47
ufl.algorithms.apply_derivatives (module), 48
ufl.algorithms.apply_function_pullbacks (module), 52
ufl.algorithms.apply_geometry_lowering (module), 52
ufl.algorithms.apply_integral_scaling (module), 53
ufl.algorithms.apply_restrictions (module), 53
ufl.algorithms.compute_form_data (module), 55
ufl.algorithms.compute_form_dependencies (module), 55
ufl.algorithms.change_to_reference (module), 57
ufl.algorithms.check_arities (module), 58
ufl.algorithms.check_restrictions (module), 59
ufl.algorithms.checks (module), 59
ufl.algorithms.computer_form_data (module), 59
ufl.algorithms.domain_analysis (module), 60
ufl.algorithms.elementtransformations (module), 61
ufl.algorithms.estimates_degrees (module), 61
ufl.algorithms.expand_compounds (module), 64
ufl.algorithms.expand_indices (module), 64
ufl.algorithms.formdata (module), 64
ufl.algorithms.formfiles (module), 65
ufl.algorithms.fortrantransformations (module), 65
ufl.algorithms.forward_ad (module), 67
ufl.algorithms.map_intervals (module), 70
ufl.algorithms.multifunction (module), 71
ufl.algorithms.pdifffs (module), 71