1 Documentation 3
Python Module Index 191
Index 193
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 part of the FEniCS Project.

For more information, visit http://www.fenicsproject.org
1.1 Installation

UFL is normally installed as part of an installation of FEniCS. If you are using UFL as part of the FEniCS software suite, it is recommended that you follow the installation instructions for FEniCS.

To install UFL itself, read on below for a list of requirements and installation instructions.

1.1.1 Requirements and dependencies

UFL requires Python version 3.5 or later and depends on the following Python packages:

- NumPy

These packages will be automatically installed as part of the installation of UFL, if not already present on your system.

1.1.2 Installation instructions

To install UFL, download the source code from the UFL Bitbucket repository, and run the following command:

```
pip install .
```

To install to a specific location, add the `--prefix` flag to the installation command:

```
pip install --prefix=<some directory> .
```
1.2 User manual

1.2.1 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.

1.2.2 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

```python
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:

\[ a(v; w) = \sum_{k=1}^{n_z} \int_{\Omega_k} I_k^z(v; w) dx + \sum_{k=1}^{n_z} \int_{\partial \Omega_k} I_k^z(v; w) ds + \sum_{k=1}^{n_z} \int_{\Gamma_k} I_k^i(v; w) dS. \]
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

- 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 support 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 evaluates to 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:

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

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

cell = Cell("triangle")

# Triangle cell embedded in 3D space

cell = Cell("triangle", 3)

Objects for regular cells of all basic shapes are predefined:

```python
# Predefined linear cells

cell = interval
cell = triangle
cell = tetrahedron
cell = quadrilateral
cell = hexahedron
```

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

## 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).
- "Bubble", representing bubble elements, useful for example to build the mini elements.
- "Quadrature" or "Q", representing artificial “finite elements” with degrees of freedom being function evaluations at quadrature points;
- "Boundary Quadrature" or "BQ", representing artificial “finite elements” with degrees of freedom being function evaluations 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 from some family on a given cell with a certain polynomial degree. Valid families and cells are explained above. The notation is

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

Some examples:

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

### 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

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

Some examples:

```python
# A quadratic "P2" vector element on a triangle
element = VectorElement("CG", triangle, 2)
# A linear 3D vector element on a 1D interval
element = VectorElement("CG", interval, 1, size=3)
# A six-dimensional piecewise constant element on a tetrahedron
element = VectorElement("DG", tetrahedron, 0, size=6)
```

### 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 :math:`(d, d)` where :math:`d` is the geometric dimension. The notation is

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

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

Examples:

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

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

Shorthand notation for two subelements:

```python
element = element1 * element2
```

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

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

represents \((e1 * e2) * e3\), i.e. this is a mixed element with two sub-elements \((e1 * e2)\) and \(e3\).

See *Form arguments* for details on how defining functions on mixed spaces can differ from defining functions on other finite element spaces.

**Examples:**

```python
# 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

```python
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):

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

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:

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

For a 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:

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

This is equivalent to the previous use of Arguments:

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

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:

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

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:

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

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)
```

There is a shorthand for this, whose use is similar to `Arguments`, called `Coefficients`:

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

Spatially constant values 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:

```python
n = FacetNormal(cell)
g = Coefficient(VectorElement("CG", cell, 1))
M = dot(n, g)*ds
```

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

```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 $j$ has been declared as a free `Index`:

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

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

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

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:

$$v = v_k \cdot i_k \equiv \sum_k v_k i_k.$$ 

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

$$A = A_{ij} i_i j_j \equiv \sum_i \sum_j A_{ij} i_i j_j.$$
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 \( i \) 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 \( v_i \) 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 \texttt{Index} objects, the indexing operator \((v[i])\) and the function \texttt{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 \texttt{Index} represents an index used for subscripting derivatives or taking components of non-scalar expressions. To create indices you can either make a single one using \texttt{Index()} or make several at once conveniently using \texttt{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:
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. $\text{rank}(A[:,i]) == 1$
- $A[i,j]$: column access, representing some column $j$ of $A$, i.e. $\text{rank}(A[:,j]) == 1$
- $C[...,0]$: subtensor access, representing the subtensor of $A$ with the last axis fixed, e.g., $A[...,0] == A[:,0]$
- $C[j,...]$: subtensor access, representing the subtensor of $A$ with the first axis fixed, e.g., $A[j,...] == A[j:]$

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 $\mathbf{v}_k$ to $\mathbf{v}$ or $A_{ij}$ to $A$, the following examples show how this is done:

```python
vk = Identity(cell.geometric_dimension())[0,k]
v = as_tensor(vk, (k,))
Aij = v[i]*u[j]
A = as_tensor(Aij, (i,j))
```

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

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

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

Here $\mathbf{v}$, $A$ and $B$ will represent the expressions

\[
\mathbf{v} = \mathbf{i}_0 + 2\mathbf{i}_1 + 3\mathbf{i}_2,
\]
\[
A = \begin{bmatrix}
  u_0 & 0 \\
  0 & u_1
\end{bmatrix},
\]
\[
B = \begin{bmatrix}
  0 & 1 \\
  1 & 2
\end{bmatrix}.
\]
Note that the function `as_tensor` generalizes from vectors to tensors of arbitrary rank, while the alternative functions `as_vector` and `as_matrix` work the same way but are only for constructing vectors and matrices. They are included for readability and convenience.

**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:

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

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:

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

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 \(+, -, *, /\) can be used freely on UFL expressions. They do have some requirements on their operands, summarized here:

**Addition or subtraction**, \( a + b \) or \( a - b \):

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

**Division**, \( a / b \):

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

**Multiplication**, \( a * b \):

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

**Basic nonlinear functions**

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

- \( \text{abs}(f) \): the absolute value of \( f \).
Unified Form Language (UFL) Documentation, Release 2019.2.0.dev0

- `sign(f)`: the sign of f (+1 or -1).
- `pow(f, g)` or `f**g`: f to the power g, \( f^g \)
- `sqrt(f)`: square root, \( \sqrt{f} \)
- `exp(f)`: exponential of f
- `ln(f)`: natural logarithm of f
- `cos(f)`: cosine of f
- `sin(f)`: sine of f
- `tan(f)`: tangent of f
- `cosh(f)`: hyperbolic cosine of f
- `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 \( f1/f2 \)
- `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:

\[
\begin{align*}
AT &= \text{transpose}(A) \\
AT &= A.T \\
AT &= \text{as_matrix}(A[i,j], (j,i))
\end{align*}
\]

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:

\[
AT = \text{as_tensor}(A[i,j,k,l], (l,k,j,i))
\]
The trace of a matrix $A$ is the sum of the diagonal entries. This can be written as:

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

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

\[
\# General tensors
f = \text{dot}(a, b)
\]

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

\[
\# Matrices a and b
f = \text{as_matrix}(a[i,k]*b[k,j], (i,j))
\]

The definition of the dot product of unit vectors is (assuming an orthonormal basis for a Euclidean space):

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

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

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

An example with a tensor of rank two

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

This is the same as a matrix-matrix multiplication.

An example with a vector and a tensor of rank two

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

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. Complex values are supported by UFL taking the complex conjugate of the second operand. This has no impact if the values are real.

If A and B are rank two tensors and C and D are rank 3 tensors their inner products are

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

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

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

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

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

Note that, in the UFL notation, dot and inner products are not equivalent for complex values.

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

\[ C \otimes D = C^* \otimes i_{r-1} \otimes i_{r-2} \otimes \cdots \otimes i_1 \otimes i_0 \]

For consistency with the inner product, the complex conjugate is taken of the first operand.

Some examples with vectors and matrices are easier to understand:

\[ v \otimes u = v^*_i u_j i_j, \]
\[ v \otimes B = v^*_i B_{ki} i_k i_l, \]
\[ A \otimes B = A^*_{ij} B_{ki} i_j i_k i_l. \]

The outer product of vectors is often written simply as

\[ v \otimes u = vu, \]

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

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

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

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

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

**det**

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

\[
d = \det(A)
\]

**dev**

The deviatoric part of matrix \( A \) can be written as

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

The definition is

\[
\text{dev}A = A - \frac{A_{ii}}{d} I
\]

where \( d \) is the rank of matrix \( A \) and \( I \) is the identity matrix.

**sym**

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

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

The definition is

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

**skew**

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

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

The definition is

\[
\text{skew}A = \frac{1}{2}(A - A^T)
\]
The cofactor of a matrix $A$ can be written as

$$B = \text{cofac}(A)$$

The definition is

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

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.

The inverse of matrix $A$ can be written as

$$A_{inv} = \text{inv}(A)$$

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
q = Dx(v[i], i)
q = v[i].dx(i)
```
Here $g$ will represent the sum of derivatives w.r.t. $x_i$ for all $i$, that is

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

**Note:** $v[i].dx(i)$ and $v_{i,i}$ 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:

- $Df = \text{grad}(f)$
- $df = \text{div}(f)$
- $cf = \text{curl}(v)$
- $rf = \text{rot}(f)$

The operand $f$ can have no free indices.

**Gradient**

The gradient of a scalar $u$ is defined as

$$\text{grad}(u) \equiv \nabla u = \sum_{k=0}^{d-1} \frac{\partial u}{\partial x_k} i_k,$$

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

The gradient of a vector $v$ is defined as

$$\text{grad}(v) \equiv \nabla v = \frac{\partial v_i}{\partial x_j} i_{ij},$$

which, written componentwise, reads

$$A = \nabla v, \quad A_{ij} = v_{i,j}$$

In general for a tensor $A$ of rank $r$ the definition is

$$\text{grad}(A) \equiv \nabla A = (\frac{\partial}{\partial x_i})(A, i_{i_0} \otimes \cdots \otimes i_{r-1}) \otimes i_i = \frac{\partial A_i}{\partial x_i} i_{i_0} \otimes \cdots \otimes i_{r-1} \otimes i_i,$$

where $i$ is a multi-index of length $r$.

In UFL, the following pairs of declarations are equivalent:

- $Df[i] = \text{grad}(f)[i]$  
- $Df = f.dx(i)$
- $Dvi[i, j] = \text{grad}(v)[i, j]$  
- $Dvi = v[i].dx(j)$
- $DAi[\ldots, i] = \text{grad}(A)[\ldots, i]$  
- $DAi = A.dx(i)$

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

$$\text{div}(v) \equiv \nabla \cdot v = \sum_{k=0}^{d-1} \frac{\partial v_i}{\partial x_i}$$

In UFL, the following declarations are equivalent:

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

for a vector expression $v$ and a tensor expression $A$.

Curl and rot

The operator `curl` or `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 `curl` or `rot` operator is only defined for vectors of length three.

In UFL, the following declarations are equivalent:

```python
omega = curl(v)
omega = rot(v)
```

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:

```python
# 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 = w**2

# The derivative of expression F w.r.t. the variable w
dF = diff(F, w)  # == 2*w
```
Note that the variable $w$ still represents the same expression. This can be useful for example to implement material laws in hyperelasticity where the stress tensor is derived from a Helmholtz strain energy function.

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

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

### Functional derivatives

The third and final kind of derivative are derivatives of functionals or forms w.r.t. a Coefficient. This is described in more detail in the section 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.

#### Restriction: $v(\text{'}+\text{'})$ and $v(\text{'}-\text{'})$

When integrating over interior facets ($\ast\text{dS}$), one may restrict expressions to the positive or negative side of the facet:

```python
element = FiniteElement("Discontinuous Lagrange", tetrahedron, 0)
v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)
a = f(\text{'}+\text{'})*dot(grad(v)(\text{'}+\text{'}), grad(u)(\text{'}-\text{'}))*\text{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: $\text{jump}(v)$

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

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

$$\text{jump}(v) \leftrightarrow [[v]] = v^+ - v^-$$

If the expression $v$ is scalar, then $\text{jump}(v)$ will also be scalar, and if $v$ is vector-valued, then $\text{jump}(v)$ will also be vector-valued.
If called with two arguments, \( \text{jump}(v, n) \) 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 \( \text{jump}(v, n) \) is given by

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

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

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

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

**Average:** \( \text{avg}(v) \)

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

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

The expression \( \text{avg}(v) \) 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:

\[
c = \text{conditional}(\text{condition}, \text{true_value}, \text{false_value})
\]

evaluates to \( \text{true_value} \) at run-time if \( \text{condition} \) evaluates to \( \text{true} \), or to \( \text{false_value} \) otherwise.

This corresponds to the C++ syntax \( (\text{condition} ? \text{true_value}: \text{false_value}) \), or the Python syntax \( (\text{true_value if condition else false_value}) \).

**Conditions**

- \( \text{eq}(a, b) \) must be used in place of the notation \( a == b \)
- \( \text{ne}(a, b) \) must be used in place of the notation \( a != b \)
- \( \text{le}(a, b) \) is equivalent to \( a <= b \)
- \( \text{ge}(a, b) \) is equivalent to \( a >= b \)
- \( \text{lt}(a, b) \) is equivalent to \( a < b \)
- \( \text{gt}(a, b) \) 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 $\epsilon$ of linear elasticity, defined by

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

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)
```

Complex values

UFL supports the definition of forms over either the real or the complex field. Indeed, UFL does not explicitly define whether `Coefficient` or `Constant` are real or complex. This is instead a matter for the form compiler to define. The complex-valued finite element spaces supported by UFL always have a real basis but complex coefficients. This means that `Constant` are `Coefficient` are complex-valued, but `Argument` is real-valued.

Complex operators

- $\text{conj}(f)$ :: complex conjugate of $f$.
- $\text{imag}(f)$ :: imaginary part of $f$.
- $\text{real}(f)$ :: real part of $f$.

Sesquilinearity

`inner` and `outer` are sesquilinear rather than linear when applied to complex values. Consequently, forms with two arguments are also sesquilinear in this case. UFL adopts the convention that inner products take the complex conjugate of the second operand. This is the usual convention in complex analysis but the reverse of the usual convention in physics.

Complex values and conditionals

Since the field of complex numbers does not admit a well order, complex expressions are not permissible as operands to `lt`, `gt`, `le`, or `ge`. When compiling complex forms, the preprocessing stage of a compiler will attempt to prove that the relevant operands are real and will raise an exception if it is unable to do so. The user may always explicitly use `real` (or `imag`) in order to ensure that the operand is real.

Compiling real forms

When the compiler treats a form as real, the preprocessing stage will discard all instances of `conj` and `real` in the form. Any instances of `imag` or complex literal constants will cause an exception.
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 dictionary. This can be used for example to replace a Coefficient with a fixed value for optimized run-time evaluation.

Example:

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

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

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:

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

To give a concrete example, these declarations are equivalent:

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

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 $|v|_A = v^T A v$ of a matrix $A$ assembled from a form $a$ can be computed with:

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

which is equivalent to:

```python
f = action(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

Sometimes 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 separating 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 \( f(w) = \int_{\Omega} \frac{1}{2} w^2 \, dx \))

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

which is equivalent to

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

Assume in the following examples that

```python
v = TestFunction(element)
u = TrialFunction(element)
w = Coefficient(element)
```

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

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

which is equivalent to

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

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

```ufl
g = Coefficient(element)
f = inner(grad(w), grad(w))*dx
F = derivative(f, w, v) + dot(w-g,v)*dx
J = derivative(F, w, u)
```
Derivatives can also be computed w.r.t. functions in mixed spaces. Consider this example, an implementation of the harmonic map equations using automatic differentiation:

```python
X = VectorElement("Lagrange", cell, 1)
Y = FiniteElement("Lagrange", cell, 1)
x = Coefficient(X)
y = Coefficient(Y)
L = inner(grad(x), grad(x))*dx + dot(x,x)*y*dx
F = derivative(L, (x,y))
J = derivative(F, (x,y))
```

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

\[
M = X + Y
\]

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

```python
u = Coefficient(M)
x, y = split(u)
L = inner(grad(x), grad(x))*dx + dot(x,x)*y*dx
F = derivative(L, u)
J = derivative(F, u)
```

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

**Combining form transformations**

Form transformations can be combined freely. Note that, to do this, derivatives are usually 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.

### 1.2.3 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 $\text{dx}$ has been declared as an integral over the interior of $\Omega$ 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)
```

(continues on next page)
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:

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

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,

\[
\begin{align*}
a(v, u) &= \int_{\Omega} \nabla v : \nabla u \; dx, \\
L(v; f) &= \int_{\Omega} v \cdot f \; dx,
\end{align*}
\]

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:

\[
\begin{align*}
a &= \text{Dx}(v[i], j) \cdot \text{Dx}(u[i], j) \cdot dx \\
L &= v[i] \cdot f[i] \cdot dx
\end{align*}
\]

or like this:

\[
\begin{align*}
a &= v[i].dx(j) \cdot u[i].dx(j) \cdot dx \\
L &= v[i] \cdot f[i] \cdot dx
\end{align*}
\]

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,

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

\( \epsilon \) being the strain.
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:

```python
a = 0.25*(Dx(v[j], i) + Dx(v[i], j))*dx
    * (Dx(u[j], i) + Dx(u[i], j))*dx
```
or like this:

```python
a = 0.25*(v[j].dx(i) + v[i].dx(j))*dx
    * (u[j].dx(i) + u[i].dx(j))*dx
```
This example is implemented in the file `Elasticity.ufl` in the collection of demonstration forms included with the UFL source distribution.

### 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:

```python
element = VectorElement("Lagrange", tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
w = Coefficient(element)
a = dot(grad(u)*w, v)*dx
```
alternatively using index notation like this:

```python
a = v[i]*w[j]*Dx(u[i], j)*dx
```
or like this:

```python
a = v[i]*w[j]*u[i].dx(j)*dx
```
This example is implemented in the file `NavierStokes.ufl` in the collection of demonstration forms included with the UFL source distribution.
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 \( u^{n-1}_h = u_h(\cdot, t_{n-1}) \) given such that
\[ \frac{1}{k_n} \int_{\Omega} v (u^n_h - u^{n-1}_h) \, dx + \int_{\Omega} c \nabla v \cdot \nabla u^n_h \, dx = \int_{\Omega} v f^n \, dx \]
for all test functions \( v \), where \( k_n = t_n - t_{n-1} \) 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 \( a(v, u^n_h) = L(v) \) for all \( v \), we obtain the following pair of bilinear and linear forms:
\[
\begin{align*}
a(v, u^n_h; c, k) &= \int_{\Omega} v u^n_h \, dx + k_n \int_{\Omega} c \nabla v \cdot \nabla u^n_h \, dx, \\
L(v; u^{n-1}_h, f, k) &= \int_{\Omega} v u^{n-1}_h \, dx + k_n \int_{\Omega} v f^n \, dx,
\end{align*}
\]
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*ex1*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:
\[
\begin{align*}
a((v, q), (u, p)) &= \int_{\Omega} \nabla v : \nabla u - (\nabla \cdot v) p + q (\nabla \cdot u) \, dx, \\
L((v, q); f) &= \int_{\Omega} v \cdot f \, dx.
\end{align*}
\]
Using a mixed formulation with Taylor-Hood elements, this can be implemented as follows:
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 \( u \in L^2 \):

\[
\begin{align*}
\sigma + \nabla u &= 0, \\
\nabla \cdot \sigma &= f.
\end{align*}
\]

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:

```python
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

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

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

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

```python
import ufl

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

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

f = Coefficient(DG1)
g = Coefficient(DG1)
h = 2.0*Circumradius(cell)
alpha = 1
gamma = 1

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

L = v*f*ds + v*g*ds
```

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

The Quadrature family

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

\[-\nabla \cdot (1 + u^2) \nabla u = f.\]

The linearised bilinear and linear forms for this equation,

\[
a(v, u; u_0) = \int_{\Omega} (1 + u_0^2) \nabla v \cdot \nabla u \, dx + \int_{\Omega} 2u_0u \nabla v \cdot \nabla u_0 \, dx,
\]

\[
L(v; u_0, f) = \int_{\Omega} vf \, dx - \int_{\Omega} (1 + u_0^2) \nabla v \cdot \nabla u_0 \, dx,
\]
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 serves 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 the bilinear and linear forms, we obtain

$$a(v, u) = \int_{\Omega} C \nabla v \cdot \nabla u \, dx + \int_{\Omega} 2 u_0 u \nabla v \cdot \nabla u_0 \, dx,$$

$$L(v; \sigma_0, f) = \int_{\Omega} v f \, dx - \int_{\Omega} \nabla v \cdot \sigma_0 \, dx.$$

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

v = TestFunction(element)
u = TrialFunction(element)
u0 = 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
# Tangent.ufl
element = FiniteElement("Lagrange", triangle, 1)
DG = FiniteElement("Discontinuous Lagrange", triangle, 0)

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

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

and
# Gradient.ufl

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, (1.0 + u0**2)*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 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 the table below. It is to be noted that the convergence rate is quadratic as it should be for all three methods.

Relative residuals for each approach for linear elements:

<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 the table below. Clearly, using the standard FiniteElement leads to a poor convergence whereas the 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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
More examples

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

1.2.4 Internal representation details

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

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

Structure of a form

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

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

General properties of expressions

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

operands

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

reconstruct

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

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

shape

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

free_indices

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

index_dimensions

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

str(u)

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

repr(u)

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

hash(u)

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

u == v

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

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 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.

1.2.5 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:
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

str

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

```
{ 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>>[everywhere], {})
+ { 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>>[everywhere], {})
```

repr

Accurate description of an 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)`:

```
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))), 'cell', Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), 'everywhere', {}, None),
  Integral(Product(Argument(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0, None),
  Coefficient(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1, dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 1)), '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)`:

```
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)
```

(continues on next page)
Inspecting and manipulating the expression tree

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

Traversing expressions

iter_expressions

Example usage:

```python
for e in iter_expressions(a):
    print str(e)
```

outputs:

- \( v_0 \ast v_1 \ast w_0 \)
- \( v_0 \ast w_1 \)

Transforming expressions

So far we presented algorithms meant to inspect 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:

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

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

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

p = Printer()
p.visit(a)

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

Visiting v_0 * v_1 with operands:
v_0, v_1

\((v_0^0)(v_1^1)\)

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
```

which 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 will 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 as 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 explicitly calls self.visit to do this.

**Automatic differentiation implementation**

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

First of all, we give a brief explanation of the algorithm. 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 an 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

\[ A_{ij} = \frac{db_i}{dw_j} = a(\phi_i, \phi_j; \sum_k w_k \phi_k), \quad \forall \phi_i \in V_h, \quad \forall \phi_j \in V_h, \]
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 $\frac{d}{dw_j}$, and we can differentiate the integrand expression instead, e.g.,

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

$$\frac{d}{dw_j} L(v; w) = \int_{\Omega} I_c \frac{d}{dw_j} \, dx + \int_{\partial\Omega} I_e \frac{d}{dw_j} \, ds.$$

In addition, we need that

$$\frac{dw}{dw_j} = \phi_j, \quad \forall \phi_j \in V_h,$$

which in UFL can be represented as

$$w = \text{Coefficient} (\text{element}),$$

$$v = \text{Argument} (\text{element}),$$

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

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.

**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.

**The computational graph**

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:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>/</td>
<td>_</td>
<td>/</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>_</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

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:

---

1 Linearized as in a linear datastructure, do not confuse this with automatic differentiation.
If we instead have the expression

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

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 follows:

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

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

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

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

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

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

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

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

1.3 ufl package

1.3.1 Subpackages

ufl.algorithms package

Submodules

ufl.algorithms.ad module

Front-end for AD routines.

\[
\text{ufl.algorithms.ad.expand_derivatives}(\text{form}, **\text{kwargs})
\]

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.

\[
\text{ufl.algorithms.analysis.extract_arguments}(a)
\]

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

\[
\text{ufl.algorithms.analysis.extract_arguments_and_coefficients}(a)
\]

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

\[
\text{ufl.algorithms.analysis.extract_coefficients}(a)
\]

Build a sorted list of all coefficients in a

\[
\text{ufl.algorithms.analysis.extract_constants}(a)
\]

Build a sorted list of all constants in a

\[
\text{ufl.algorithms.analysis.extract_elements}(\text{form})
\]

Build sorted tuple of all elements used in form.

\[
\text{ufl.algorithms.analysis.extract_sub_elements}(\text{elements})
\]

Build sorted tuple of all sub elements (including parent element).

\[
\text{ufl.algorithms.analysis.extract_type}(a, \text{ufl_type})
\]

Build a set of all objects of class ufl_type found in a. The argument a can be a Form, Integral or Expr.

\[
\text{ufl.algorithms.analysis.extract_unique_elements}(\text{form})
\]

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.

    ```python
    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\( (o, A) \)

\texttt{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).

\texttt{ufl.algorithms.apply\_derivatives} module

This module contains the apply\_derivatives algorithm which computes the derivatives of a form of expression.

\texttt{class ufl.algorithms.apply\_derivatives.CoordinateDerivativeRuleDispatcher}

\texttt{Bases: ufl.corealg.multifunction.MultiFunction}

\texttt{coefficient\_derivative}\( (o) \)

\texttt{coordinate\_derivative}\( (o) \)

\texttt{derivative}\( (o) \)

\texttt{expr}\( (o, *ops) \)

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{grad}\( (o) \)

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

\texttt{terminal}\( (o) \)

\texttt{class ufl.algorithms.apply\_derivatives.CoordinateDerivativeRuleset}\( (coefficients, arguments, coefficient\_derivatives) \)

\texttt{Bases: ufl.algorithms.apply\_derivatives.GenericDerivativeRuleset}

Apply AFD (Automatic Functional Differentiation) to expression.

Implements rules for the Gateaux derivative \( D_w[v](\ldots) \) defined as

\[ D_w[v](e) = \frac{d}{d\tau} e(w+\tau v)|_{\tau=0} \]

where ‘e’ is a ufl form after pullback and \( w \) is a SpatialCoordinate.

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

Return a zero with the right shape for terminals independent of differentiation variable.

\texttt{coefficient}\( (o) \)

\texttt{geometric\_quantity}\( (o) \)

Return a zero with the right shape for terminals independent of differentiation variable.

\texttt{grad}\( (o) \)

\texttt{jacobian}\( (o) \)

\texttt{reference\_grad}(g)
class ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher
Bases: ufl.corealg.multifunction.MultiFunction

coefficient_derivative(o, f, dummy_w, dummy_v, dummy_cd)
coordinate_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.

      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)

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 $D_w[v](\ldots)$ defined as

$$D_w[v](e) = \frac{d}{dtau} e(w+tau v)|_{tau=0}$$

argument(o)
  Return a zero with the right shape for terminals independent of differentiation variable.

cell_avg(o, fp)
coefficient(o)
coordinate_derivative(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(\(o, unused_{dc}, dt, df\))
conj(\(o, df\))

constant(\(o\))
Return a zero with the right shape for terminals independent of differentiation variable.

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(\(o\))
Trigger error for types with missing handlers.

facet_avg(\(o\))

fixme(\(o\))

form_argument(\(o\))
geometric_quantity(\(o\))

grad(\(o\))
imag(\(o, df\))

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** \((o, da, db)\)

**real** \((o, df)\)

**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** \((o)\)

\[
\frac{dX}{dx} = \text{inv}(\frac{dx}{dX}) = \text{inv}(J) = K
\]

**coefficient** \((o)\)

**facet_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 \(\text{Grad}(g)\). Override for specific types if other behaviour is needed.
grad(o)
    Represent grad(grad(f)) as Grad(Grad(f)).

jacobian_inverse(o)

reference_grad(o)

reference_value(o)

spatial_coordinate(o)
    \frac{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)
    \frac{dX}{dX} = I

coefficient(o)

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

generic_quantity(o)
    \frac{dg}{dX} = 0 \text{ 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)
    \frac{dx}{dX} = J

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

argument(o)
    Return a zero with the right shape for terminals independent of differentiation variable.

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

coefficient(o)
    \frac{df}{dv} = \text{Id if } v = f \text{ else } 0.
    \text{Note that if } v = \text{variable}(f), \frac{df}{dv} \text{ is still 0, but if } v = f, \text{ i.e. } \text{isinstance}(v, \text{Coefficient}) = \text{True}, \text{ then } \frac{df}{dv} = \frac{df}{df} = \text{Id}.

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

generic_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.

**reference_value**(o)

**variable**(o, df, l)

```python
ufl.algorithms.apply_derivatives.apply_coordinate_derivatives(expression)
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.

```python
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.
    ```python
    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.
```

```python
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.

```python
class ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier(preserve_types=())
    Bases: ufl.corealg.multifunction.MultiFunction
    cell_coordinate(o)
    cell_diameter(o)
    cell_normal(o)
    cell_volume(o)
```

---

**1.3. ufl package**
circumradius \( (o) \)

\textbf{expr} \( (o, \ast\text{ops}) \)

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.

\textbf{facet_area} \( (o) \)

\textbf{facet_cell_coordinate} \( (o) \)

\textbf{facet_jacobian} \( (o) \)

\textbf{facet_jacobian_determinant} \( (o) \)

\textbf{facet_jacobian_inverse} \( (o) \)

\textbf{facet_normal} \( (o) \)

\textbf{jacobian} \( (o) \)

\textbf{jacobian_determinant} \( (o) \)

\textbf{jacobian_inverse} \( (o) \)

\textbf{max_cell_edge_length} \( (o) \)

\textbf{max_facet_edge_length} \( (o) \)

\textbf{min_cell_edge_length} \( (o) \)

\textbf{min_facet_edge_length} \( (o) \)

\textbf{spatial_coordinate} \( (o) \)

\textbf{terminal} \( (t) \)

\begin{verbatim}
uf1.algorithms.apply_geometry_lowering.apply_geometry_lowering(form, preserve_types=())
\end{verbatim}

Change GeometricQuantity objects in expression to the lowest level GeometricQuantity objects.

Assumes the expression is preprocessed or at least that derivatives have been expanded.

@ \textbf{param form}: An Expr or Form.

\textbf{ufl.algorithms.apply_integral_scaling module}

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

\begin{verbatim}
uf1.algorithms.apply_integral_scaling.apply_integral_scaling(form)
\end{verbatim}

Multiply integrands by a factor to scale the integral to reference frame.

\begin{verbatim}
uf1.algorithms.apply_integral_scaling.compute_integrand_scaling_factor(integral)
\end{verbatim}

Change integrand geometry to the right representations.

\textbf{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.DefaultRestrictionApplier(side=None)
Bases: ufl.corealg.multifunction.MultiFunction

derivative(o)

facet_area(o)
Restrict a continuous quantity to default side if no current restriction is set.

facet_jacobian(o)
Restrict a continuous quantity to default side if no current restriction is set.

facet_jacobian_determinant(o)
Restrict a continuous quantity to default side if no current restriction is set.

facet_jacobian_inverse(o)
Restrict a continuous quantity to default side if no current restriction is set.

facet_origin(o)
Restrict a continuous quantity to default side if no current restriction is set.

max_facet_edge_length(o)
Restrict a continuous quantity to default side if no current restriction is set.

min_facet_edge_length(o)
Restrict a continuous quantity to default side if no current restriction is set.

operator(o, *ops)
Reuse object if operands are the same objects.
Use in your own subclass by setting e.g.

```python
expr = MultiFunction.reuse_if_un touched
```
as a default rule.

restricted(o)

spatial_coordinate(o)
Restrict a continuous quantity to default side if no current restriction is set.

terminal(o)

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.

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

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

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

facet_normal(o)

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

gemetric_facet_quantity(o)
Restrict a discontinuous quantity to current side, require a side to be set.
**grad**, *(o)*

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

**label**, *(o)*

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

**multi_index**, *(o)*

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

**operator**, *(o, *ops)*

Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```python
expr = MultiFunction.reuse_if_unmerged
```

as a default rule.

**quadrature_weight**, *(o)*

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

**reference_cell_volume**, *(o)*

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

**reference_facet_volume**, *(o)*

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

**reference_value**, *(o)*

Reference value of something follows same restriction rule as the underlying object.

**restricted**, *(o)*

When hitting a restricted quantity, visit child with a separate restriction algorithm.

**terminal**, *(o)*

**variable**, *(o, op, label)*

Strip variable.

`ufl.algorithms.apply_restrictions.apply_default_restrictions(expression)`

Some terminals can be restricted from either side.

This applies a default restriction to such terminals if unrestricted.

`ufl.algorithms.apply_restrictions.apply_restrictions(expression)`

Propagate restriction nodes to wrap differential terminals directly.

**ufl.algorithms.balancing module**

```python
class ufl.algorithms.balancing.BalanceModifiers
    Bases: ufl.corealg.multifunction.MultiFunction

    cell_avg(expr, *ops)
    expr(expr, *ops)
        Trigger error for types with missing handlers.
    facet_avg(expr, *ops)
    grad(expr, *ops)
    negative Restricted(expr, *ops)
    positive Restricted(expr, *ops)
```
\begin{verbatim}

reference_grad(expr, *ops)
reference_value(expr, *ops)
terminal(expr)
ufl.algorithms.balancing.balance_modified_terminal(expr)
ufl.algorithms.balancing.balance_modifiers(expr)

\end{verbatim}

\textbf{ufl.algorithms.change_to_reference module}

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

\textbf{class} ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad
\textbf{Bases:} ufl.corealg.multifunction.MultiFunction

\begin{verbatim}
cell_avg(o, *dummy_ops)
coefficient_derivative(o, *dummy_ops)
expr(o, *ops)
    Trigger error for types with missing handlers.
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)
\end{verbatim}

\textbf{class} ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad
\textbf{Bases:} ufl.corealg.multifunction.MultiFunction

\begin{verbatim}
coefficient_derivative(o)
expr(o, *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.
grad(o)
reference_grad(o)
terminal(o)
\end{verbatim}

1.3. ufl package 57
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.AarityChecker(arguments)
Bases: ufl.corealg.multifunction.MultiFunction

    argument(o)
    cell_avg(o, a)
    component_tensor(o, a, i)
    conditional(o, c, a, b)
    conj(o, a)
    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)
```python
terminal(o)
variable(o, f, l)
```

**exception** `ufl.algorithms.check_arities.ArityMismatch`

Bases: `ufl.log.UFLException`

```python
ufl.algorithms.check_arities.check_form_arity(form, arguments, complex_mode=False)
ufl.algorithms.check_arities.check_integrand_arity(expr, arguments, complex_mode=False)
```

**ufl.algorithms.check_restrictions module**

Algorithms related to restrictions.

```python
class ufl.algorithms.check_restrictions.RestrictionChecker(require_restriction)
Bases: ufl.corealg.multifunction.MultiFunction
```

- `expr(o)`
  - Trigger error for types with missing handlers.
- `facet_normal(o)`
- `form_argument(o)`
- `restricted(o)`

```python
ufl.algorithms.check_restrictions.check_restrictions(expression, require_restriction)
```

Check that types that must be restricted are restricted in expression.

**ufl.algorithms.checks module**

Functions to check the validity of forms.

```python
ufl.algorithms.checks.validate_form(form)
```

Performs all implemented validations on a form. Raises exception if something fails.

**ufl.algorithms.comparison_checker module**

Algorithm to check for ‘comparison’ nodes in a form when the user is in ‘complex mode’

```python
class ufl.algorithms.comparison_checker.CheckComparisons
Bases: ufl.corealg.multifunction.MultiFunction
```

- Raises an error if comparisons are done with complex quantities.
- If quantities are real, adds the Real operator to the compared quantities.
- Terminals that are real are `RealValue`, `Zero`, and `Argument` (even in complex FEM, the basis functions are real)
- Operations that produce reals are `Abs`, `Real`, `Imag`. Terminals default to complex, and `Sqrt`, `Pow` (defensively) imply complex. Otherwise, operators preserve the type of their operands.
- `abs(o, *ops)`
- `compare(o, *ops)`
```
expr(o, *ops)
    Defaults expressions to complex unless they only act on real quantities. Overridden for specific operators.
    Rebuilds objects if necessary.

ge(o, *ops)
gt(o, *ops)
imag(o, *ops)
le(o, *ops)
lt(o, *ops)
max_value(o, *ops)
min_value(o, *ops)
power(o, base, exponent)
real(o, *ops)
sign(o, *ops)
sqrt(o, *ops)
terminal(term, *ops)

exception ufl.algorithms.comparison_checker.ComplexComparisonError
    Bases: Exception

ufl.algorithms.comparison_checker.do_comparison_check(form)
    Raises an error if invalid comparison nodes exist

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.attach_estimated_degrees(form)
    Attach estimated polynomial degree to a form’s integrals.

    Parameters
    form -- The Form to inspect.

    Returns
    A new Form with estimate degrees attached.

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_default_restrictions=True,
    do_apply_restrictions=True,
    do_estimate_degrees=True,
    do_append_everywhere_integrals=True,
    complex_mode=False)

ufl.algorithms.coordinate_derivative_helpers module

This module provides the necessary tools to strip away and then reattach the coordinate derivatives at the right time point in compute_form_data.
class ufl.algorithms.coordinate_derivative_helpers.CoordinateDerivativeIsOutermostChecker
    Bases: ufl.corealg.multifunction.MultiFunction

    Traverses the tree to make sure that CoordinateDerivatives are only on the outside. The visitor returns False as long as no CoordinateDerivative has been seen.

    coordinate_derivative (o, expr, *)
    expr (o, *operands)
        If we have already seen a CoordinateDerivative, then no other expressions apart from more CoordinateDerivatives are allowed to wrap around it.

    multi_index (o)
    terminal (o)

ufl.algorithms.coordinate_derivative_helpers.attach_coordinate_derivatives (integral, coordinate_derivatives)

ufl.algorithms.coordinate_derivative_helpers.strip_coordinate_derivatives (integrals)

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)
Build integral data given a list of integrals.

**Parameters**
- **integrals** – An iterable of *Integral* objects.

**Returns**
- A tuple of *IntegralData* objects.

The integrals you pass in here must have been rearranged and gathered (removing the “everywhere” subdomain_id. To do this, you should call `group_form_integrals()`.

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

ufl.algorithms.domain_analysis.group_form_integrals(form, domains, do_append_everywhere_integrals=True)
Group integrals by domain and type, performing canonical simplification.

**Parameters**
- **form** – the *Form* to group the integrals of.
- **domains** – an iterable of :class:`~.Domain`’s.

**Returns**
- A new *Form* with gathered integrands.

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, do_append_everywhere_integrals=True)
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

This module provides helper functions to - FFC/DOLFIN adaptive chain, - UFL algorithms taking care of underspecified DOLFIN expressions.

ufl.algorithms.elementtransformations.increase_order(element)
Return element of same family, but a polynomial degree higher.

ufl.algorithms.elementtransformations.tear(element)
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.IrreducibleInt

**Bases:** int

Degree type used by quadrilaterals.
Unlike int, values of this type are not decremented by \_reduce\_degree.

```python
class ufl.algorithms.estimate_degrees.SumDegreeEstimator(default_degree, element_replace_map)
Bases: ufl.corealg.multifunction.MultiFunction
```

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

**abs**(\(v, a\))
- This is a heuristic, correct if there is no argument\(v\).

**argument**(\(v\))
- A form argument provides a degree depending on the element, or the default degree if the element has no degree.

**atan\_2**(\(v, a, b\))
- 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.

**bessel\_function**(\(v, nu, x\))
- 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.

**cell\_avg**(\(v, a\))
- Cell average of a function is always cellwise constant.

**cell\_coordinate**(\(v\))
- A coordinate provides one additional degree.

**coefficient**(\(v\))
- A form argument provides a degree depending on the element, or the default degree if the element has no degree.

**cofactor**(\(v, *args\))

**component\_tensor**(\(v, A, ii\))

**compound\_derivative**(\(v, *args\))

**compound\_tensor\_operator**(\(v, *args\))

**condition**(\(v, *args\))

**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.

**conj**(\(v, a\))

**constant**(\(v\))

**constant\_value**(\(v\))
- Constant values are constant.

**coordinate\_derivative**(\(v, integrand\_degree, b, direction\_degree, d\))
- We use the heuristic that a shape derivative in direction \(V\) introduces terms \(V\) and grad(\(V\)) into the integrand. Hence we add the degree of the deformation to the estimate.

**cross**(\(v, *ops\))

**curl**(\(v, f\))
- Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when TensorProduct elements or quadrilateral elements are involved.
derivative(v, *args)

determinant(v, *args)

deviatoric(v, *args)

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

division(v, *ops)
    Using the sum here is a heuristic. Consider e.g. (x+1)/(x-1).

dot(v, *ops)

expr(v, *ops)
    For most operators we take the max degree of its operands.

eexpr_list(v, *o)

eexpr_mapping(v, *o)

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 TensorProduct elements or quadrilateral elements are involved.

imag(v, a)

index_sum(v, A, ii)

indexed(v, A, ii)

inner(v, *ops)

inverse(v, *args)

label(v)

list_tensor(v, *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 TensorProduct elements or quadrilateral elements are involved.

nabla_grad(v, f)
    Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when TensorProduct elements or quadrilateral elements are involved.
negative_restricted\((v, a)\)

outer\((v, \*ops)\)

positive_restricted\((v, a)\)

power\((v, a, b)\)

If \(b\) is a positive integer: degree\((a^b)\) == degree\((a)\)*\(b\) otherwise use the heuristic degree\((a^b)\) == degree\((a)\) + 2

product\((v, \*ops)\)

real\((v, a)\)

reference_curl\((v, f)\)

Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when TensorProduct elements or quadrilateral elements are involved.

reference_div\((v, f)\)

Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when TensorProduct elements or quadrilateral elements are involved.

reference_grad\((v, f)\)

Reduces the estimated degree by one; used when derivatives are taken. Does not reduce the degree when TensorProduct elements or quadrilateral 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\((v, \*args)\)

\[\text{ufl.algorithms.estimate_degrees.estimate_total_polynomial_degree}(e, \text{ default_degree}=1, \text{element_replace_map}={}())\]

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.

\[\text{ufl.algorithms.expand_compounds module}\]

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

\[\text{ufl.algorithms.expand_compounds.expand_compounds}(e)\]
ufl.algorithms.expand_indices module

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

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

    component ()
        Return current component tuple.

    component_tensor (x)

    conditional (x)

    division (x)

    form_argument (x)

    grad (x)

    index_sum (x)

    indexed (x)

    list_tensor (x)

    multi_index (x)

    scalar_value (x)

    terminal (x)
        Always reuse Expr (ignore children)

    zero (x)

ufl.algorithms.expand_indices.expand_indices (e)

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.

class ufl.algorithms.formdata.ExprData
    Bases: object

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

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.
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_lines_decoded(fn)
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(lines)
    Replace ‘#include foo.ufl’ statements with contents of foo.ufl.

ufl.algorithms.formsplitter module

Extract part of a form in a mixed FunctionSpace.

class ufl.algorithms.formsplitter.FormSplitter
    Bases: ufl.corealg.multifunction.MultiFunction
    argument (obj)
    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.
    multi_index (obj)
    split (form, ix, iy=0)
ufl.algorithms.formsplitter.extract_blocks (form, i=None, j=None)

ufl.algorithms.formtransformations module

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

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.

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

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 $\geq 0$, providing whatever Argument its operand does.

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

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

list_tensor ($x$, *ops)

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

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

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

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\((\text{form, coefficient})\)

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\((\text{form, coefficient})\)

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\((\text{form}, \text{reordered_arguments=None})\)

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\((\text{form})\)

Return set of arities of terms present in form.

ufl.algorithms.formtransformations.compute_form_functional\((\text{form})\)

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\((\text{form})\)

Compute the left hand side of a form.

Example:

\[
a = u^*v^*dx + f^*v^*dx \quad a = \text{lhs}(a) -> u^*v^*dx
\]

ufl.algorithms.formtransformations.compute_form_rhs\((\text{form})\)

Compute the right hand side of a form.

Example:

\[
a = u^*v^*dx + f^*v^*dx \quad L = \text{rhs}(a) -> -f^*v^*dx
\]

ufl.algorithms.formtransformations.compute_form_with_arity\((\text{form}, \text{arity}, \text{arguments=None})\)

Compute parts of form of given arity.
ufl.algorithms.formtransformations.zero_expr(e)

**ufl.algorithms.map_integrands module**

Basic algorithms for applying functions to subexpressions.

```python
ufl.algorithms.map_integrands.map_integrand_dags(function, form, only_integral_type=None, compress=True)
```

```python
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.

**ufl.algorithms.multifunction module**

**ufl.algorithms.remove_complex_nodes module**

Algorithm for removing conj, real, and imag nodes from a form for when the user is in ‘real mode’

```python
class ufl.algorithms.remove_complex_nodes.ComplexNodeRemoval
    Bases: ufl.corealg.multifunction.MultiFunction
    Replaces complex operator nodes with their children
    conj(o, a)
    expr(o, *ops)
    imag(o, a)
    real(o, a)
    terminal(t, *ops)
```

```python
ufl.algorithms.remove_complex_nodes.remove_complex_nodes(expr)
```

Replaces complex operator nodes with their children. This is called during compute_form_data if the compiler wishes to compile real-valued forms. In essence this strips all trace of complex support from the preprocessed form.

**ufl.algorithms.renumbering module**

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

```python
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)
```
class ufl.algorithms.renumbering.VariableRenumberingTransformer
   Bases: ufl.algorithms.transformer.ReuseTransformer
   variable (o)
   ufl.algorithms.renumbering.renumber_indices(expr)

ufl.algorithms.replace 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.
      ```python
      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, *operands)
      Always reconstruct expr.
class ufl.algorithms.transformer.ReuseTransformer (variable_cache=None)
Bases: ufl.algorithms.transformer.Transformer

eexpr (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)
    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, *operands)
    Always reconstruct expr.

eexpr (o)
    Trigger error.

print_visit_stack ()

reconstruct_variable (o)

reuse (o)
    Always reuse Expr (ignore children)

reuse_if_possible (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.

reuse_if_untouched (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.

reuse_variable (o)

terminal (o)
    Always reuse Expr (ignore children)

undefined (o)
    Trigger error.
class ufl.algorithms.transformer.VariableStripper
    Bases: ufl.algorithms.transformer.ReuseTransformer

def visit(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.

def is_post_handler(function):
    Is this a handler that expects transformed children as input?

def strip_variables(e):
    Replace all Variable instances with the expression they represent.

def 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.

def 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.

def 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.

def 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.

def 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.computer_form_data(form, do_apply_function_pullbacks=False,
    do_apply_integral_scaling=False,
    do_apply_geometry_lowering=False,
    preserve_geometry_types=(),
    do_apply_default_restrictions=True,
    do_apply_restrictions=True, do_estimate_degrees=True,
    do_append_everywhere_integrals=True, complex_mode=False)

ufl.algorithms.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.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.

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

    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)
Always reuse Expr (ignore children)

variable(o)

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

class ufl.algorithms.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, *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, *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*, *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.

**reuse_variable** (*o*)

**terminal** (*o*)
Always reuse Expr (ignore children)

**undefined** (*o*)
Trigger error.

**visit** (*o*)

```python
class MultiFunction
    Bases: object
    Base class for collections of non-recursive 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 Python’s function call overhead 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.

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

```python
ufl.algorithms.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.
```

```python
ufl.algorithms.extract_elements (form)
Build sorted tuple of all elements used in form.
```

```python
ufl.algorithms.extract_sub_elements (*elements*)
Build sorted tuple of all sub elements (including parent element).
```

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

```python
ufl.algorithms.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.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.extract_coefficients(a)
    Build a sorted list of all coefficients in a, which can be a Form, Integral or Expr.

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

ufl.algorithms.post_traversal(expr)
    Yield o for each node o in expr, child before parent.

ufl.algorithms.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.expand_compounds(e)

ufl.algorithms.validate_form(form)
    Performs all implemented validations on a form. Raises exception if something fails.

class ufl.algorithms.FormSplitter
    Bases: ufl.corealg.multifunction.MultiFunction

    argument(obj)

    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.

    multi_index(obj)

    split(form, ix, iy=0)

ufl.algorithms.extract_arguments(a)
    Build a sorted list of all arguments in a, which can be a Form, Integral or Expr.

ufl.algorithms.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.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.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.

```python
ufl.algorithms.compute_form_lhs(form)
Compute the left hand side of a form.

Example:
\[ a = u^*v^*dx + f^*v^*dx \]
\[ a = \text{lhs}(a) \rightarrow u^*v^*dx \]
```

```python
ufl.algorithms.compute_form_rhs(form)
Compute the right hand side of a form.

Example:
\[ a = u^*v^*dx + f^*v^*dx \]
\[ L = \text{rhs}(a) \rightarrow -f^*v^*dx \]
```

```python
ufl.algorithms.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?)
```

```python
ufl.algorithms.compute_form_signature(form, renumbering)
```

```python
ufl.algorithms.tree_format(expression, indentation=0, parentheses=True)
```

### ufl.finiteelement package

#### Submodules

**ufl.finiteelement.brokenelement module**

```python
class ufl.finiteelement.brokenelement.BrokenElement(element)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
The discontinuous version of an existing Finite Element space.

mapping()
Not implemented.

reconstruct(**kwargs)

shortstr()
Format as string for pretty printing.
```

**ufl.finiteelement.elementlist 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.

```python
ufl.finiteelement.elementlist.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 ved input data against the element list and aliases defined in ufl.
```

1.3. ufl package
**ufl.finiteelement.elementlist.feec_element (family, n, r, k)**

Finite element exterior calculus notation n = topological dimension of domain r = polynomial order k = form_degree

**ufl.finiteelement.elementlist.feec_element_l2 (family, n, r, k)**

Finite element exterior calculus notation n = topological dimension of domain r = polynomial order k = form_degree

**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.

**ufl.finiteelement.elementlist.register_element2 (family, value_rank, sobolev_space, mapping, degree_range, cellnames)**

Register new finite element family.

**ufl.finiteelement.elementlist.show_elements ()**

Shows all registered elements.

---

**ufl.finiteelement.enrichedelement module**

This module defines the UFL finite element classes.

**class** ufl.finiteelement.enrichedelement.EnrichedElement (*elements*)

Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

The vector sum of several finite element spaces:

\[
\text{EnrichedElement}(V, Q) = \{ v + q | v \in V, q \in Q \}.
\]

Dual basis is a concatenation of subelements dual bases; primal basis is a concatenation of subelements primal bases; resulting element is not nodal even when subelements are. Structured basis may be exploited in form compilers.

**is_cellwise_constant ()**

Return whether the basis functions of this element is spatially constant over each cell.

**shortstr ()**

Format as string for pretty printing.

**class** ufl.finiteelement.enrichedelement.EnrichedElementBase (*elements*)

Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The vector sum of several finite element spaces:

\[
\text{EnrichedElement}(V, Q) = \{ v + q | v \in V, q \in Q \}.
\]

**mapping ()**

Not implemented.

**reconstruct (**kwargs**)**

**sobolev_space ()**

Return the underlying Sobolev space.
class ufl.finiteelement.enrichedelement.NodalEnrichedElement(*elements)
    Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

The vector sum of several finite element spaces:

\[ \text{EnrichedElement}(V, Q) = \{ v + q | v \in V, q \in Q \}. \]

Primal basis is reorthogonalized to dual basis which is a concatenation of subelements dual bases; resulting element is nodal.

is_cellwise_constant()
    Return whether the basis functions of this element is spatially constant over each cell.

shortstr()
    Format as string for pretty printing.

ufl.finiteelement.facetelement module

ufl.finiteelement.facetelement.FacetElement(element)
    Constructs the restriction of a finite element to the facets of the cell.

ufl.finiteelement.finiteelement module

This module defines the UFL finite element classes.

class ufl.finiteelement.finiteelement.FiniteElement(family, cell=None, degree=None, form_degree=None, quad_scheme=None, variant=None)
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The basic finite element class for all simple finite elements.

mapping()
    Not implemented.

reconstruct(family=None, cell=None, degree=None, form_degree=None, quad_scheme=None, variant=None)
    Construct a new FiniteElement object with some properties replaced with new values.

shortstr()
    Format as string for pretty printing.

sobolev_space()
    Return the underlying Sobolev space.

variant()

ufl.finiteelement.finiteelementbase module

This module defines the UFL finite element classes.

class ufl.finiteelement.finiteelementbase.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()  
  Not implemented.

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 $c_0 \rightarrow c_1$ meaning that component $c_0$ is represented by component $c_1$. A component is a tuple of one or more ints.

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.finiteelementbase.FiniteElementBase

  A curl-conforming version of an outer product element, assuming this makes mathematical sense.
mapping()  
Not implemented.

reconstruct(**kwargs)

shortstr()  
Format as string for pretty printing.

sobolev_space()  
Return the underlying Sobolev space.

class ufl.finiteelement.hdivcurl.HDivElement(element)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

A div-conforming version of an outer product element, assuming this makes mathematical sense.

mapping()  
Not implemented.

reconstruct(**kwargs)

shortstr()  
Format as string for pretty printing.

sobolev_space()  
Return the underlying Sobolev space.

ufl.finiteelement.interiorelement module

ufl.finiteelement.interiorelement.InteriorElement(element)

Constructs the restriction of a finite element to the interior of the cell.

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** *(component=None)*
Return whether the basis functions of this element is spatially constant over each cell.

**mapping**
Not implemented.

**num_sub_elements**
Return number of sub elements.

**reconstruct** (**kwargs**)
**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 \( c_0 \rightarrow c_1 \) 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** *(family, cell=None, degree=None, 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**
Not implemented.

**reconstruct** (**kwargs**)
**shortstr**
Format as string for pretty printing.

**symmetry**
Return the symmetry dict, which is a mapping \( c_0 \rightarrow c_1 \) 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.VectorElement** *(family, cell=None, degree=None, 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.

**reconstruct** (**kwargs**)
**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.

is_cellwise_constant()
   Return whether the basis functions of this element is spatially constant over each cell.

mapping()
   Not implemented.

num_restricted_sub_elements()
   Return number of restricted sub elements.

num_sub_elements()
   Return number of sub elements.

reconstruct(**kwargs)

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 $c_0 \rightarrow c_1$ meaning that component $c_0$ is represented by component $c_1$. A component is a tuple of one or more ints.

ufl.finiteelement.tensorproductelement module

This module defines the UFL finite element classes.

class ufl.finiteelement.tensorproductelement.TensorProductElement(*elements, **kwargs)

Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The tensor product of $d$ element spaces:

$$V = V_1 \otimes V_2 \otimes \ldots \otimes V_d$$

Given bases $\{\phi_{j_i}\}$ of the spaces $V_i$ for $i = 1, \ldots, d$, $\{\phi_{j_1} \otimes \phi_{j_2} \otimes \cdots \otimes \phi_{j_d}\}$ forms a basis for $V$.

mapping()
   Not implemented.

num_sub_elements()
   Return number of subelements.

reconstruct(cell=None)

shortstr(cell=None)
   Short pretty-print.
sobolev_space()
    Return the underlying Sobolev space of the TensorProductElement.

sub_elements()
    Return subelements (factors).

Module contents

This module defines the UFL finite element classes.

1.3.2 Submodules

1.3.3 ufl.algebra module

Basic algebra operations.

class ufl.algebra.Abs(a)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.algebra.Conj(a)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.algebra.Division(a, b)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape = ()

class ufl.algebra.Imag(a)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
    ufl_free_indices
    ufl_index_dimensions
ufl_shape

class ufl.algebra.Power(a, b)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.

def ufl_free_indices

def ufl_index_dimensions

def 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)
        Evaluate expression at given coordinate with given values for terminals.

def ufl_free_indices

def ufl_index_dimensions

def ufl_shape = ()

class ufl.algebra.Real(a)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.

def ufl_free_indices

def ufl_index_dimensions

def ufl_shape

class ufl.algebra.Sum(a, b)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.

def ufl_free_indices

def ufl_index_dimensions

def ufl_shape

1.3.4 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.
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
number()
  Return the Argument number.

part()

ufl_domain()
  Deprecated, please use .ufl_function_space().ufl_domain() instead.

ufl_domains()
  Deprecated, please use .ufl_function_space().ufl_domains() instead.

ufl_element()
  Deprecated, please use .ufl_function_space().ufl_element() instead.

ufl_function_space()
  Get the function space of this Argument.

ufl_shape
  Return the associated UFL shape.

ufl.argument.Arguments (function_space, number)
  UFL value: Create an Argument in a mixed space, and return a tuple with the function components correspond-
  ing to the subelements.

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 corre-
  sponding 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 corre-
  sponding to the subelements.

1.3.5 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.

1.3.6 ufl.averaging module

Averaging operations.

class ufl.averaging.CellAvg(f)
  Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index_values)\)
   Performs an approximate symbolic evaluation, since we dont have a cell.

\texttt{ufl_free_indices}
\texttt{ufl_index_dimensions}
\texttt{ufl_shape}

\texttt{class ufl.averaging.FacetAvg(f)}
\texttt{Bases: ufl.core.operator.Operator}
\texttt{evaluate \((x, mapping, component, index_values)\)}
   Performs an approximate symbolic evaluation, since we dont have a cell.

\texttt{ufl_free_indices}
\texttt{ufl_index_dimensions}
\texttt{ufl_shape}

1.3.7 \texttt{ufl.cell} module

Types for representing a cell.

\texttt{class ufl.cell.AbstractCell(topological_dimension, geometric_dimension)}
\texttt{Bases: object}
   Representation of an abstract finite element cell with only the dimensions known.

\texttt{geometric_dimension()}
   Return the dimension of the space this cell is embedded in.

\texttt{has_simplex_facets()}
   Return True if all the facets of this cell are simplex cells.

\texttt{is_simplex()}
   Return True if this is a simplex cell.

\texttt{topological_dimension()}
   Return the dimension of the topology of this cell.

\texttt{class ufl.cell.Cell(cellname, geometric_dimension=None)}
\texttt{Bases: ufl.cell.AbstractCell}
   Representation of a named finite element cell with known structure.

\texttt{cellname()}
   Return the cellname of the cell.

\texttt{has_simplex_facets()}
   Return True if all the facets of this cell are simplex cells.

\texttt{is_simplex()}
   Return True if this is a simplex cell.

\texttt{num_edges()}
   The number of cell edges.

\texttt{num_facet_edges()}
   The number of facet edges.

\texttt{num_facets()}
   The number of cell facets.
num_vertices()
    The number of cell vertices.
reconstruct (geometric_dimension=None)
class ufl.cell.TensorProductCell(*cells, **kwargs)
    Bases: ufl.cell.AbstractCell
cellname()
    Return the cellname of the cell.
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.
reconstruct (geometric_dimension=None)
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.

1.3.8 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.

1.3.9 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 Expr instance will have certain properties. The most important ones are 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. Type traits are accessed via a class or instance object of 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::.

    @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

    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
    Transpose a rank-2 tensor expression. For more general transpose operations of higher order tensor expressions, use indexing and Tensor.

dx(*ii)
    Return the partial derivative with respect to spatial variable number ii.

evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.
geometric_dimension()
    Return the geometric dimension this expression lives in.

static ufl_disable_profiling()
    Turn off the object counting mechanism. Return 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 the 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
    An abstract class for a form argument.

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

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.

    count()
evaluate \((x, \text{mapping}, \text{component}, \text{index\_values})\)
Return the value of the coordinate.

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

name = 'x'
ufl\_shape
Return the number of coordinates defined (i.e. the geometric dimension of the domain).

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
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

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
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

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()
Return whether this expression is spatially constant over each cell (or over each facet for facet quantities).
**name** = 'x0'

**ufl_shape**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

```python
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**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

```python
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**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

```python
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**

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

```python
class ufl.classes.FacetJacobian(domain)
Bases: ufl.geometry.GeometricFacetQuantity
```

UFL geometry representation: The Jacobian of the mapping from reference facet to spatial coordinates.

\[ FJ_{ij} = dx_i/dXf_j \]

The FacetJacobian is the product of the Jacobian and CellFacetJacobian:
\[
FJ = \frac{dx}{dX_f} = \frac{dx}{dX} \frac{dX}{dX_f} = J \cdot CFJ
\]

`is_cellwise_constant()`

Return whether this expression is spatially constant over each cell.

`name = 'FJ'`

`ufl_shape`

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

class ufl.classes.CellFacetJacobian(domain)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The Jacobian of the mapping from reference facet to reference cell coordinates.

\[
CFJ_{ij} = \frac{dX_i}{dX_f} \frac{dX_f_j}{dX_f}
\]

`is_cellwise_constant()`

Return whether this expression is spatially constant over each cell.

`name = 'CFJ'`

`ufl_shape`

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

class ufl.classes.ReferenceCellEdgeVectors(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 = 'RCEV'`

`ufl_shape`

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

class ufl.classes.ReferenceFacetEdgeVectors(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 = 'RFEV'`

`ufl_shape`

Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized
from iterable’s items.

If the argument is a tuple, the return value is the same object.

class ufl.classes.CellVertices (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: Physical cell vertices.
   is_cellwise_constant ()
      Return whether this expression is spatially constant over each cell.
   name = 'CV'
   ufl_shape
      Built-in immutable sequence.
      If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized
      from iterable’s items.
      If the argument is a tuple, the return value is the same object.

class ufl.classes.CellEdgeVectors (domain)
   Bases: ufl.geometry.GeometricCellQuantity
   UFL geometry representation: The vectors between physical cell vertices for each edge in cell.
   is_cellwise_constant ()
      Return whether this expression is spatially constant over each cell.
   name = 'CEV'
   ufl_shape
      Built-in immutable sequence.
      If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized
      from iterable’s items.
      If the argument is a tuple, the return value is the same object.

class ufl.classes.FacetEdgeVectors (domain)
   Bases: ufl.geometry.GeometricFacetQuantity
   UFL geometry representation: The vectors between physical 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
      Built-in immutable sequence.
      If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized
      from iterable’s items.
      If the argument is a tuple, the return value is the same object.

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

Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The pseudo-determinant of the FacetJacobian.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'detFJ'**

class ufl.classes.CellFacetJacobianDeterminant

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.classes.JacobianInverse

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**

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

class ufl.classes.FacetJacobianInverse

Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The pseudo-inverse of the FacetJacobian.

**is_cellwise_constant()**

Return whether this expression is spatially constant over each cell.

**name = 'FK'**

**ufl_shape**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

class ufl.classes.CellFacetJacobianInverse

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**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

```python
class ufl.classes.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
Return the number of coordinates defined (i.e. the geometric dimension of the domain).
```

```python
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
Return the number of coordinates defined (i.e. the geometric dimension of the domain).
```

```python
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
Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.
```

```python
class ufl.classes.ReferenceCellVolume(domain)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The volume of the reference cell.

name = 'reference_cell_volume'
```

```python
class ufl.classesREFERENCE FACET VOLUME (domain)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The volume of the reference cell of the current facet.

name = 'reference_facet_volume'
```

```python
class ufl.classes.CellVolume(domain)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The volume of the cell.

name = 'volume'
```
class ufl.classes.Circumradius\(\text{(domain)}\)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The circumradius of the cell.
name = 'circumradius'

class ufl.classes.CellDiameter\(\text{(domain)}\)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The diameter of the cell, i.e., maximal distance of two points in the cell.
name = 'diameter'

class ufl.classes.FacetArea\(\text{(domain)}\)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The area of the facet.
name = 'facetarea'

class ufl.classes.MinCellEdgeLength\(\text{(domain)}\)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The minimum edge length of the cell.
name = 'mincelledgelength'

class ufl.classes.MaxCellEdgeLength\(\text{(domain)}\)
Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The maximum edge length of the cell.
name = 'maxcelledgelength'

class ufl.classes.MinFacetEdgeLength\(\text{(domain)}\)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The minimum edge length of the facet.
name = 'minfacetedgelength'

class ufl.classes.MaxFacetEdgeLength\(\text{(domain)}\)
Bases: ufl.geometry.GeometricFacetQuantity

UFL geometry representation: The maximum edge length of the facet.
name = 'maxfacetedgelength'

class ufl.classes.CellOrientation\(\text{(domain)}\)
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\(\text{(domain)}\)
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

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

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)
    Evaluate index.

indices()
    Return tuple of indices.

is_cellwise_constant()
    Always True.

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

ufl_free_indices
    This shall not be used.

ufl_index_dimensions
    This shall not be used.

ufl_shape
    This shall not be used.

class ufl.classes.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.classes.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)
    Get self from mapping and return the component asked for.

ufl_free_indices
    Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

**ufl_index_dimensions**
Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

**ufl_shape**

```python
class ufl.classes.ScalarValue(value)
    Bases: ufl.constantvalue.ConstantValue
    A constant scalar value.

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

imag()
real()

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()
value()
```

**class ufl.classes.ComplexValue(value)**
Bases: ufl.constantvalue.ScalarValue

UFL literal type: Representation of a constant, complex scalar

**argument()**

**modulus()**

```python
class ufl.classes.RealValue(value)
    Bases: ufl.constantvalue.ScalarValue
    Abstract class used to differentiate real values from complex ones

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()

class ufl.classes.FloatValue(value)
    Bases: ufl.constantvalue.RealValue
    UFL literal type: Representation of a constant scalar floating point value.

class ufl.classes.IntValue(value)
    Bases: ufl.constantvalue.RealValue
    UFL literal type: Representation of a constant scalar integer value.

class ufl.classes.Identity(dim)
    Bases: ufl.constantvalue.ConstantValue
```

1.3. ufl package
UFL literal type: Representation of an identity matrix.

\texttt{evaluate}(x, \textit{mapping}, \textit{component}, \textit{index\_values})

Evaluates the identity matrix on the given components.

\texttt{ufl\_shape}

\texttt{class ufl.classes.PermutationSymbol}(\textit{dim})

Bases: \texttt{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.

\texttt{evaluate}(x, \textit{mapping}, \textit{component}, \textit{index\_values})

Evaluates the permutation symbol.

\texttt{ufl\_shape}

\texttt{class ufl.classes.Indexed}(\textit{expression}, \textit{multiindex})

Bases: \texttt{ufl.core.operator.Operator}

\texttt{evaluate}(x, \textit{mapping}, \textit{component}, \textit{index\_values}, \textit{derivatives=}())

Evaluate expression at given coordinate with given values for terminals.

\texttt{ufl\_free\_indices}

\texttt{ufl\_index\_dimensions}

\texttt{ufl\_shape = ()}

\texttt{class ufl.classes.ListTensor}(\*\textit{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, \textit{mapping}, \textit{component}, \textit{index\_values}, \textit{derivatives=}())

Evaluate expression at given coordinate with given values for terminals.

\texttt{ufl\_free\_indices}

\texttt{ufl\_index\_dimensions}

\texttt{ufl\_shape}

\texttt{class ufl.classes.ComponentTensor}(\textit{expression}, \textit{indices})

Bases: \texttt{ufl.core.operator.Operator}

UFL operator type: Maps the free indices of a scalar valued expression to tensor axes.

\texttt{evaluate}(x, \textit{mapping}, \textit{component}, \textit{index\_values})

Evaluate expression at given coordinate with given values for terminals.

\texttt{indices()}  

\texttt{ufl\_free\_indices}

\texttt{ufl\_index\_dimensions}

\texttt{ufl\_shape}

\texttt{class ufl.classes.Argument}(\textit{function\_space}, \textit{number}, \textit{part=None})

Bases: \texttt{ufl.core.terminal.FormArgument}

UFL value: Representation of an argument to a form.

\texttt{is\_cellwise\_constant()}  

Return whether this expression is spatially constant over each cell.
number()  
Return the Argument number.

part()  
ufl_domain()  
Deprecated, please use .ufl_function_space().ufl_domain() instead.

ufl_domains()  
Deprecated, please use .ufl_function_space().ufl_domains() instead.

ufl_element()  
Deprecated, please use .ufl_function_space().ufl_element() instead.

ufl_function_space()  
Get the function space of this Argument.

ufl_shape  
Return the associated 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()  
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  
Return the associated UFL shape.

class ufl.classes.Constant(domain, shape=(), count=None)  
Bases: ufl.core.terminal.Terminal  
count()  
is_cellwise_constant()  

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

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

ufl_shape  

class ufl.classes.Label(count=None)  
Bases: ufl.core.terminal.Terminal  
count()  
is_cellwise_constant()  

1.3. ufl package
**ufl_domains()**
Return tuple of domains related to this terminal object.

**ufl_free_indices**
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

**ufl_index_dimensions**
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

**ufl_shape**

```python
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:
```
```python
e = <...>
e = variable(e)
f = exp(e**2)
df = diff(f, e)
```

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

**expression()**

**label()**

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

**ufl_free_indices** = ()
**ufl_index_dimensions** = ()
**ufl_shape**

```python
class ufl.classes.Sum(a, b)
    Bases: ufl.core.operator.Operator
    The product of two or more UFL objects.
```

```python
class ufl.classes.Product(a, b)
    Bases: ufl.core.operator.Operator
```

**evaluate**(x, mapping, component, index_values)
Evaluate expression at given coordinate with given values for terminals.
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape = ()

class ufl.classes.Division \((a, b)\)
Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape = ()

class ufl.classes.Power \((a, b)\)
Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape = ()

class ufl.classes.Abs \((a)\)
Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape

class ufl.classes.Conj \((a)\)
Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape

class ufl.classes.Real \((a)\)
Bases: ufl.core.operator.Operator
evaluate \((x, mapping, component, index\_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape

1.3. ufl package
class ufl.classes.Imag(a)
    Bases: ufl.core.operator.Operator
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
    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
    ufl_index_dimensions
    ufl_shape

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
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

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)
        Evaluate expression at given coordinate with given values for terminals.
    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)
        Evaluate expression at given coordinate with given values for terminals.
    
    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.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()
        Return all domains this expression is defined on.
    
    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.CoordinateDerivative(integrand, coefficients, arguments, coefficient_derivatives)

Bases: ufl.differentiation.CoefficientDerivative

Derivative of the integrand of a form w.r.t. the SpatialCoordinates.

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.CompoundDerivative(operands)

Bases: ufl.differentiation.Derivative

Base class for all compound derivative types.

class ufl.classes.Grad(f)

Bases: ufl.differentiation.CompoundDerivative

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.CompoundDerivative

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.CompoundDerivative

ufl_free_indices
ufl_index_dimensions
ufl_shape

class ufl.classes.ReferenceDiv(f)

Bases: ufl.differentiation.CompoundDerivative

ufl_free_indices
ufl_index_dimensions
ufl_shape
class ufl.classes.NablaGrad(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.NablaDiv(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.Curl(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.classes.ReferenceCurl(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

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)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.NE(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.LE(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.
class ufl.classes.GE(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.LT(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.GT(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.AndCondition(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.OrCondition(left, right)
    Bases: ufl.conditional.BinaryCondition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.NotCondition(condition)
    Bases: ufl.conditional.Condition
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.Conditional(condition, true_value, false_value)
    Bases: ufl.core.operator.Operator
evaluate(x, mapping, component, index_values)
    Evaluate expression at given coordinate with given values for terminals.

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)
    Evaluate expression at given coordinate with given values for terminals.

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 \((x, mapping, component, index_values)\)
Evaluate expression at given coordinate with given values for terminals.

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()

class ufl.classes.MathFunction\(\text{name, argument}\)
Bases: ufl.core.operator.Operator
Base class for all unary scalar math functions.

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

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()

class ufl.classes.Sqrt\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Exp\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Ln\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

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

class ufl.classes.Cos\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Sin\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Tan\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Cosh\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Sinh\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Tanh\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Acos\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Asin\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Atan\(\text{argument}\)
Bases: ufl.mathfunctions.MathFunction

class ufl.classes.Atan2\(\text{arg1, arg2}\)
Bases: ufl.core.operator.Operator

evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
   Evaluate expression at given coordinate with given values for terminals.

ufl\_free\_indices = ()
ufl\_index\_dimensions = ()
ufl\_shape = ()

class ufl.classes.Erf(argument)
   Bases: ufl.mathfunctions.MathFunction
   evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
   Evaluate expression at given coordinate with given values for terminals.

class ufl.classes.BesselFunction(name, classname, nu, argument)
   Bases: ufl.core.operator.Operator
   Base class for all bessel functions
   evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
   Evaluate expression at given coordinate with given values for terminals.

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.CellAvg(f)
   Bases: ufl.core.operator.Operator
   evaluate \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
    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 \( (x, \text{mapping}, \text{component}, \text{index\_values}) \)
    Performs an approximate symbolic evaluation, since we dont have a cell.

ufl\_free\_indices
ufl\_index\_dimensions
ufl\_shape
```python
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()
    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_facet_edges()
    The number of facet edges.
    num_facets()
    The number of cell facets.
    num_vertices()
    The number of cell vertices.
    reconstruct(geometric_dimension=None)

class ufl.classes.TensorProductCell(*cells, **kwargs)
    Bases: ufl.cell.AbstractCell
    cellname()
    Return the cellname of the cell.
```

112 Chapter 1. Documentation
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.

reconstruct(geometric_dimension=None)

sub_cells()
   Return list of cell factors.

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 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()
   Not implemented.

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 \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

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, variant=None)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The basic finite element class for all simple finite elements.

mapping()
    Not implemented.

reconstruct (family=None, cell=None, degree=None, quad_scheme=None, variant=None)
    Construct a new FiniteElement object with some properties replaced with new values.

shortstr()
    Format as string for pretty printing.

sobolev_space()
    Return the underlying Sobolev space.

variant()

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 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 (component=None)
    Return whether the basis functions of this element is spatially constant over each cell.

mapping()
    Not implemented.
num_sub_elements()
    Return number of sub elements.

reconstruct (**kwargs)

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 \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.classes.VectorElement (family, cell=None, degree=None, 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.

reconstruct (**kwargs)

shortstr()
    Format as string for pretty printing.

class ufl.classes.TensorElement (family, cell=None, degree=None, 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()
    Not implemented.

reconstruct (**kwargs)

shortstr()
    Format as string for pretty printing.

symmetry()
    Return the symmetry dict, which is a mapping \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.classes.EnrichedElement (*elements)
    Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

    The vector sum of several finite element spaces:
    
    \[
    \text{EnrichedElement}(V, Q) = \{ v + q | v \in V, q \in Q \}.
    \]

    Dual basis is a concatenation of subelements dual bases; primal basis is a concatenation of subelements primal bases; resulting element is not nodal even when subelements are. Structured basis may be exploited in form compilers.

is_cellwise_constant()
    Return whether the basis functions of this element is spatially constant over each cell.
shortstr()
Format as string for pretty printing.

class ufl.classes.NodalEnrichedElement(*elements)
Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

The vector sum of several finite element spaces:

\[ \text{EnrichedElement}(V, Q) = \{ v + q | v \in V, q \in Q \} \].

Primal basis is reorthogonalized to dual basis which is a concatenation of subelements dual bases; resulting element is nodal.

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

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.

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

mapping()
Not implemented.

num_restricted_sub_elements()
Return number of restricted sub elements.

num_sub_elements()
Return number of sub elements.

reconstruct(**kwargs)

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 \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.classes.TensorProductElement(*elements, **kwargs)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The tensor product of \( d \) element spaces:

\[ V = V_1 \otimes V_2 \otimes \ldots \otimes V_d \]

Given bases \( \{ \phi_j \} \) of the spaces \( V_i \) for \( i = 1, \ldots, d \), \( \{ \phi_{j_1} \otimes \phi_{j_2} \otimes \cdots \otimes \phi_{j_d} \} \) forms a basis for \( V \).
mapping()  
    Not implemented.

num_sub_elements()  
    Return number of subelements.

reconstruct(cell=None)  

shortstr()  
    Short pretty-print.

sobolev_space()  
    Return the underlying Sobolev space of the TensorProductElement.

sub_elements()  
    Return subelements (factors).

class ufl.classes.HDivElement(element)  
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    A div-conforming version of an outer product element, assuming this makes mathematical sense.

    mapping()  
        Not implemented.

    reconstruct(**kwargs)

    shortstr()  
        Format as string for pretty printing.

    sobolev_space()  
        Return the underlying Sobolev space.

class ufl.classes.HCurlElement(element)  
    Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

    A curl-conforming version of an outer product element, assuming this makes mathematical sense.

    mapping()  
        Not implemented.

    reconstruct(**kwargs)

    shortstr()  
        Format as string for pretty printing.

    sobolev_space()  
        Return the underlying Sobolev space.

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

    The discontinuous version of an existing Finite Element space.

    mapping()  
        Not implemented.

    reconstruct(**kwargs)

    shortstr()  
        Format as string for pretty printing.

ufl.classes.FacetElement(element)  
    Constructs the restriction of a finite element to the facets of the cell.
ufl.classes.InteriorElement(element) Constructs the restriction of a finite element to the interior of the cell.

class ufl.classes.AbstractDomain(topological_dimension, geometric_dimension) Bases: object Symbolic representation of a geometric domain with only a geometric and topological dimension.

generic_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.

is_piecewise_linear_simplex_domain() ufl_cargo() Return carried object that will not be used by UFL.

ufl_cell() ufl_coordinate_element()

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()
    Return ufl domain.

ufl_domains()
    Return ufl domains.

ufl_element()
    Return ufl element.

ufl_sub_spaces()
    Return ufl sub spaces.

class ufl.classes.MixedFunctionSpace(*args)
    Bases: ufl.functionspace.AbstractFunctionSpace

    num_sub_spaces()
    ufl_domain()
        Return ufl domain.
    ufl_domains()
        Return ufl domains.
    ufl_element()
    ufl_elements()
        Return ufl elements.
    ufl_sub_space(i)
        Return i-th ufl sub space.
    ufl_sub_spaces()
        Return ufl sub spaces.

class ufl.classes.TensorProductFunctionSpace(*function_spaces)
    Bases: ufl.functionspace.AbstractFunctionSpace

    ufl_sub_spaces()

class ufl.classes.IndexBase
    Bases: object
    Base class for all indices.

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()

ufl.classes.TestFunction(function_space, part=None)
    UFL value: Create a test function argument to a form.

ufl.classes.TrialFunction(function_space, part=None)
    UFL value: Create a trial function argument to a form.
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.

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

    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.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.

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 \( \quad b = a.\text{reconstruct}() \quad c = a.\text{reconstruct}() \)

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.

coefficient_numbering()
Return a contiguous numbering of coefficients in a mapping \{coefficient: number\}.

coefficients()
Return all Coefficient objects found in form.

constants()

domain_numbering()
Return a contiguous numbering of domains in a mapping \{domain: number\}.

empty()
Returns whether the form has no integrals.

equals(other)
Evaluate \( \text{bool}(\text{lhs\_form} == \text{rhs\_form}) \).

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_domain(domain)
Return a sequence of all integrals with a particular integration domain.

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.

class ufl.classes.Equation(lhs, rhs)
Bases: object
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.

1.3.10 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()
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**
Return the associated 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.

### 1.3.11 ufl.compound_expressions module

Functions implementing compound expressions as equivalent representations using basic operators.

- `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.
1.3.12 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)
        Evaluate expression at given coordinate with given values for terminals.

class ufl.conditional.BinaryCondition(name, left, right):
    Bases: ufl.conditional.Condition

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.conditional.Condition(operands):
    Bases: ufl.core.operator.Operator

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_shape = ()

class ufl.conditional.Conditional(condition, true_value, false_value):
    Bases: ufl.core.operator.Operator

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

    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.conditional.EQ(left, right):
    Bases: ufl.conditional.BinaryCondition

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

class ufl.conditional.GE(left, right):
    Bases: ufl.conditional.BinaryCondition

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

class ufl.conditional.GT(left, right):
    Bases: ufl.conditional.BinaryCondition

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

class ufl.conditional.LE(left, right):
    Bases: ufl.conditional.BinaryCondition

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

class ufl.conditional.LT(left, right):
    Bases: ufl.conditional.BinaryCondition

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

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)
        Evaluate expression at given coordinate with given values for terminals.

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()
class ufl.conditional.NE(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
class ufl.conditional.NotCondition(condition)
    Bases: ufl.conditional.Condition
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.
class ufl.conditional.OrCondition(left, right)
    Bases: ufl.conditional.BinaryCondition
    evaluate(x, mapping, component, index_values)
        Evaluate expression at given coordinate with given values for terminals.

1.3.13 ufl.constant module

This module defines classes representing non-literal values which are constant with respect to a domain.
class ufl.constant.Constant(domain, shape=(), count=None)
    Bases: ufl.core.terminal.Terminal
    count()
is_cellwise_constant()
ufl_domain()
    Return the single unique domain this expression is defined on, or throw an error.
ufl_domains()
    Return tuple of domains related to this terminal object.
ufl_shape
1.3.14 ufl.constantvalue module

This module defines classes representing constant values.

class ufl.constantvalue.ComplexValue(value)
    Bases: ufl.constantvalue.ScalarValue
    UFL literal type: Representation of a constant, complex scalar
        argument()
        modulus()

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.RealValue
    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, mapping, component, index_values)
            Evaluates the identity matrix on the given components.

    ufl_shape

class ufl.constantvalue.IntValue(value)
    Bases: ufl.constantvalue.RealValue
    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, mapping, component, index_values)
            Evaluates the permutation symbol.

    ufl_shape

class ufl.constantvalue.RealValue(value)
    Bases: ufl.constantvalue.ScalarValue
    Abstract class used to differentiate real values from complex ones
        ufl_free_indices = ()
        ufl_index_dimensions = ()
```python
ufl_shape = ()

class ufl.constantvalue.ScalarValue(value)
    Bases: ufl.constantvalue.ConstantValue
    A constant scalar value.
    evaluate(x, mapping, component, index_values)
    Get self from mapping and return the component asked for.

imag()
real()

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape = ()
value()

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)
    Get self from mapping and return the component asked for.

ufl_free_indices
    Built-in immutable sequence.
    If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
    If the argument is a tuple, the return value is the same object.

ufl_index_dimensions
    Built-in immutable sequence.
    If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
    If the argument is a tuple, the return value is the same object.

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.

1.3.15 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.

```python
class ufl.differentiation.CompoundDerivative(operands)

Bases: ufl.differentiation.Derivative

Base class for all compound derivative types.
```
class ufl.differentiation.NablaGrad(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.differentiation.ReferenceCurl(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.differentiation.ReferenceDiv(f)
    Bases: ufl.differentiation.CompoundDerivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

class ufl.differentiation.ReferenceGrad(f)
    Bases: ufl.differentiation.CompoundDerivative
    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.differentiation.VariableDerivative(f, v)
    Bases: ufl.differentiation.Derivative
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

1.3.16 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.
    `is_piecewise_linear_simplex_domain()`
    `ufl_cargo()`
        Return carried object that will not be used by UFL.
    `ufl_cell()`
    `ufl_coordinate_element()`
    `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.

### 1.3.17 ufl.equation module

The Equation class, used to express equations like a == L.

class ufl.equation.Equation(lhs, rhs)
Bases: object

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.

### 1.3.18 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()
- index_dimensions()
- ufl_domains()
  Return all domains this expression is defined on.
- ufl_free_indices
- ufl_index_dimensions
- ufl_shape

### 1.3.19 ufl.exprequals module

ufl.exprequals.exprequals(expr, 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(expr, 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.

1.3. ufl package
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.

1.3.20 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

1.3.21 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.

    coefficient_numbering()
        Return a contiguous numbering of coefficients in a mapping {coefficient:number}.

    coefficients()
        Return all Coefficient objects found in form.

    constants()

    domain_numbering()
        Return a contiguous numbering of domains in a mapping {domain:number}.

    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_domain(domain)
        Return a sequence of all integrals with a particular integration domain.
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.

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, sometimes occurring in pydolfin, e.g. assemble(1*dx, mesh=mesh).

ufl.form.sub_forms_by_domain(form)
Return a list of forms each with an integration domain

1.3.22 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, and taking the complex conjugate of the result.

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=None, 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 Coefficient s 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.formoperators.extract_blocks(form, i=None, j=None)
ufl.formoperators.functional(form)
ufl.formoperators.lhs(form)
ufl.formoperators.rhs(form)
ufl.formoperators.sensitivity_rhs(a, u, L, v)
```

**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.extract_blocks(form, i=None, j=None)**
UFL form operator: Given a linear or bilinear form on a mixed space, extract the block corresponding to the indices \(i, j\).

Example:

\[
a = \text{inner} \left( \text{grad}(u), \text{grad}(v) \right) \text{dx} + \text{div}(u) \text{q} \text{dx} + \text{div}(v) \text{p} \text{dx}
\]

\[
\text{extract_blocks}(a, 0, 0) \rightarrow \text{inner} \left( \text{grad}(u), \text{grad}(v) \right) \text{dx}
\]

**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:

\[
a = uv \text{dx} + fv \text{dx}
a = \text{lhs}(a) \rightarrow uv \text{dx}
\]

**ufl.formoperators.rhs(form)**
UFL form operator: Given a combined bilinear and linear form, extract the right hand side (negated linear form part).

Example:

\[
a = uv \text{dx} + fv \text{dx}
L = \text{rhs}(a) \rightarrow -fv \text{dx}
\]

**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

\[
0 = \frac{d}{dv} (Ax - b) = \frac{dA}{dv} x + A \frac{dx}{dv} - \frac{db}{dv},
\]

\[
A \frac{dx}{dv} = \frac{db}{dv} - \frac{dA}{dv} x,
\]

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 = variable(v_expression)
L = IL(v) * dx
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 \( \frac{da}{dv} \) and \( \frac{dL}{dv} \) from the forms

\[
da = \text{diff}(a, v) \\
dL = \text{diff}(L, v)
\]

and the action of \( da \) on \( u \) by

\[
da u = \text{action}(da, u)
\]

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,

\[
dL = \text{sensitivity_rhs}(a, u, L, v)
\]

\[\text{ufl.formoperators.set_list_item}(li, i, v)\]
\[\text{ufl.formoperators.system}(\text{form})\]

UFL form operator: Split a form into the left hand side and right hand side, see \( \text{lhs} \) and \( \text{rhs} \).

\[\text{ufl.formoperators.zero_lists}(\text{shape})\]

**1.3.23 ufl.functionspace module**

Types for representing function spaces.

```python
class ufl.functionspace.AbstractFunctionSpace:
    Bases: object
    _ufl_sub_spaces()

class ufl.functionspace.FunctionSpace(domain, element):
    Bases: ufl.functionspace.AbstractFunctionSpace
    _ufl_domain()
        Return ufl domain.
    _ufl_domains()
        Return ufl domains.
    _ufl_element()
        Return ufl element.
    _ufl_sub_spaces()
        Return ufl sub spaces.

class ufl.functionspace.MixedFunctionSpace(*args):
    Bases: ufl.functionspace.AbstractFunctionSpace
    num_sub_spaces()
        Return ufl sub spaces.
    _ufl_domain()
        Return ufl domain.
```
ufl_domains()
Return ufl domains.

ufl_element()

ufl_elements()
Return ufl elements.

ufl_sub_space(i)
Return i-th ufl sub space.

ufl_sub_spaces()
Return ufl sub spaces.

class ufl.functionspace.TensorProductFunctionSpace(*function_spaces)
Bases: ufl.functionspace.AbstractFunctionSpace

1.3.24 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
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

class ufl.geometry.CellDiameter(domain)
Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The diameter of the cell, i.e., maximal distance of two points in the cell.

name = 'diameter'

class ufl.geometry.CellEdgeVectors(domain)
Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The vectors between physical cell vertices for each edge in cell.

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

name = 'CEV'
ufl_shape
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

class ufl.geometry.CellFacetJacobian (domain)
Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The Jacobian of the mapping from reference facet to reference cell coordinates.
CFJ\_ij = \frac{dX_i}{dX_f j}

is\_cellwise\_constant ()
Return whether this expression is spatially constant over each cell.

name = 'CFJ'
ufl_shape
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

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
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

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
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

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
       Return the number of coordinates defined (i.e. the geometric dimension of the domain).

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'

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()
       Return whether this expression is spatially constant over each cell (or over each facet for facet quantities).

   name = 'x0'

   ufl_shape
       Built-in immutable sequence.

       If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

       If the argument is a tuple, the return value is the same object.

class ufl.geometry.CellVertices(domain)
   Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: Physical cell vertices.

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

   name = 'CV'

   ufl_shape
       Built-in immutable sequence.

       If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

       If the argument is a tuple, the return value is the same object.
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.
    is_cellwise_constant()
        Return whether this expression is spatially constant over each cell.
    name = 'Xf'

    ufl_shape
        Built-in immutable sequence.
        If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
        If the argument is a tuple, the return value is the same object.

class ufl.geometry.FacetEdgeVectors(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The vectors between physical 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
        Built-in immutable sequence.
        If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
        If the argument is a tuple, the return value is the same object.

class ufl.geometry.FacetJacobian(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The Jacobian of the mapping from reference facet to spatial coordinates.
\[ F_{J,ij} = \frac{dx_i}{dXf_j} \]

The FacetJacobian is the product of the Jacobian and CellFacetJacobian:

\[ FJ = \frac{dx}{dXf} = \frac{dx}{dX} \frac{dX}{dXf} = J \cdot CFJ \]

`is_cellwise_constant()`

Return whether this expression is spatially constant over each cell.

`name = 'FJ'`

`ufl_shape`

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.

If the argument is a tuple, the return value is the same object.

```python
class ufl.geometry.FacetJacobianDeterminant(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The pseudo-determinant of the FacetJacobian.
    is_cellwise_constant()
    Return whether this expression is spatially constant over each cell.
    name = 'detFJ'
```

```python
class ufl.geometry.FacetJacobianInverse(domain)
    Bases: ufl.geometry.GeometricFacetQuantity
    UFL geometry representation: The pseudo-inverse of the FacetJacobian.
    is_cellwise_constant()
    Return whether this expression is spatially constant over each cell.
    name = 'FK'
```

```python
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'
```

```python
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
    Built-in immutable sequence.
    If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
    If the argument is a tuple, the return value is the same object.

class ufl.geometry.GeometricCellQuantity(domain)
    Bases: ufl.geometry.GeometricQuantity

class ufl.geometry.GeometricFacetQuantity(domain)
    Bases: ufl.geometry.GeometricQuantity

class ufl.geometry.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()
    Return tuple of domains related to this terminal object.

    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.

    \[ J_{ij} = \frac{dx_i}{dX_j} \]

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

    name = 'J'

    ufl_shape
    Return the number of coordinates defined (i.e. the geometric dimension of the domain).

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.

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

```python
name = 'K'
```

```python
ufl_shape
Return the number of coordinates defined (i.e. the geometric dimension of the domain).
```

```python
class ufl.geometry.MaxCellEdgeLength(domain)
Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The maximum edge length of the cell.

```python
name = 'maxcelledgelength'
```

```python
class ufl.geometry.MaxFacetEdgeLength(domain)
Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The maximum edge length of the facet.

```python
name = 'maxfacetedgelength'
```

```python
class ufl.geometry.MinCellEdgeLength(domain)
Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The minimum edge length of the cell.

```python
name = 'mincelledgelength'
```

```python
class ufl.geometry.MinFacetEdgeLength(domain)
Bases: ufl.geometry.GeometricFacetQuantity
UFL geometry representation: The minimum edge length of the facet.

```python
name = 'minfacetedgelength'
```

```python
class ufl.geometry.QuadratureWeight(domain)
Bases: ufl.geometry.GeometricQuantity
UFL geometry representation: The current quadrature weight.
Only used inside a quadrature context.

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

```python
name = 'weight'
```

```python
class ufl.geometry.ReferenceCellEdgeVectors(domain)
Bases: ufl.geometry.GeometricCellQuantity
UFL geometry representation: The vectors between reference cell vertices for each edge in cell.

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

```python
name = 'RCEV'
```

```python
ufl_shape
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.
```
class ufl.geometry.ReferenceCellVolume(domain)
    Bases: ufl.geometry.GeometricCellQuantity

    UFL geometry representation: The volume of the reference cell.

ame = 'reference_cell_volume'

class ufl.geometry.ReferenceFacetEdgeVectors(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 = 'RFEV'

ufl_shape
    Built-in immutable sequence.
    If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.
    If the argument is a tuple, the return value is the same object.

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
    Built-in immutable sequence.
    If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.
    If the argument is a tuple, the return value is the same object.

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.

count()

evaluate(x, mapping, component, index_values)
    Return the value of the coordinate.

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

name = 'x'
ufl_shape
Return the number of coordinates defined (i.e. the geometric dimension of the domain).

1.3.25 ufl.index_combination_utils module
Utilities for analysing and manipulating free index tuples

ufl.index_combination_utils.create_slice_indices(component, shape, fi)
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[i,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.

1.3.26 ufl.indexed module
This module defines the Indexed class.

class ufl.indexed.Indexed(expression, multiindex)
Bases: ufl.core.operator.Operator

evaluate(x, mapping, component, index_values, derivatives=())
Evaluate expression at given coordinate with given values for terminals.

ufl_free_indices
ufl_index_dimensions
ufl_shape = ()

1.3.27 ufl.indexsum module
This module defines the IndexSum class.

class ufl.indexsum.IndexSum(summand, index)
Bases: ufl.core.operator.Operator

dimension()

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

index()
ufl_free_indices
ufl_index_dimensions
ufl_shape

1.3.28 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.

    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.
```

1.3.29 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=<class 'Exception'>):
    Bases: object

    add_indent(increment=1)
    Add to indentation level.

    add_logfile(filename=None, mode='a', level=10)
    Add a log file.

    begin(*message)
    Begin task: write message and increase indentation level.

    debug(*message)
    Write debug message.
```

1.3. ufl package
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)
Gets the handler to the file identified by the given file name.

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 one, 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.
1.3.30 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)
        Evaluate expression at given coordinate with given values for terminals.

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_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)
        Evaluate expression at given coordinate with given values for terminals.

    ufl_free_indices = ()
    ufl_index_dimensions = ()
    ufl_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)
        Evaluate expression at given coordinate with given values for terminals.

class ufl.mathfunctions.Exp(argument)
    Bases: ufl.mathfunctions.MathFunction
class `ufl.mathfunctions.Ln`(*argument*)
Bases: `ufl.mathfunctions.MathFunction`

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

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*)
Evaluate expression at given coordinate with given values for terminals.

```python
def ufl_free_indices():  
    return ()

def ufl_index_dimensions():  
    return ()

def ufl_shape():  
    return ()
```

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`

### 1.3.31 ufl.measure module

The Measure class.

class `ufl.measure.Measure`(*integral_type, domain=None, subdomain_id='everywhere', metadata=None, subdomain_data=None*)
Bases: `object`

**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:**
```python
<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:**
```python
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
    \( f^*(d\Omega_1*d\Omega_2) \)
    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
    \( f^*(d\gamma(1)+d\gamma(3)) \)
    into
    \( f^*d\gamma(1) + f^*d\gamma(3) \)

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)

1.3.32 ufl.objects module

Utility objects for pretty syntax in user code.

1.3.33 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, \( D_n(f) := \text{dot}(\text{grad}(f), n) \).
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 sine 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 with two the arguments f1 and f2.

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.conj(f)
   UFL operator: The complex conjugate of f

ufl.operators.conjugate(f)
   UFL operator: The complex conjugate of f

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 cosine of f.

ufl.operators.cosh(f)
UFL operator: Take the hyperbolic cosine 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

\[ \text{div}(v) = v[i].dx(i) \]
\[ \text{div}(T)[:] = T[:,i].dx(i) \]

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. The complex conjugate of the second argument is taken.

ufl.operators.elem_div(A, B)
UFL operator: Take the elementwise division of tensors A and B with the same shape.

ufl.operators.elem_mult(A, B)
UFL operator: Take the elementwise multiplication of tensors A and B with the same shape.

ufl.operators.elem_op(op, *args)
UFL operator: Take the elementwise application of operator op on scalar values from one or more tensor arguments.

ufl.operators.elem_op_items(op_ind, indices, *args)

ufl.operators.elem_pow(A, B)
UFL operator: Take the elementwise power of tensors A and B with the same shape.

ufl.operators.eq(left, right)
UFL operator: A boolean expression (left == right) for use with conditional.
ufl.operators.erf(f)
UFL operator: Take the error function of f.

ufl.operators.exp(f)
UFL operator: Take the exponential of f.

ufl.operators.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.facet_avg(f)
UFL operator: Take the average of v over a facet.

ufl.operators.ge(left, right)
UFL operator: A boolean expression (left >= right) for use with conditional.

ufl.operators.grad(f)
UFL operator: Take the gradient of f.

This operator follows the grad convention where
grad(s)[i] = s.dx(i)
grad(v)[i,j] = v[i].dx(j)
grad(T)[i,:] = T[:,i].dx(i)

for scalar expressions s, vector expressions v, and arbitrary rank tensor expressions T.

See also: nabla_grad()

ufl.operators.gt(left, right)
UFL operator: A boolean expression (left > right) for use with conditional.

ufl.operators.imag(f)
UFL operator: The imaginary part of f

ufl.operators.inner(a, b)
UFL operator: Take the inner product of a and b. The complex conjugate of the second argument is taken.

ufl.operators.inv(A)
UFL operator: Take the inverse of A.

ufl.operators.jump(v, n=None)
UFL operator: Take the jump of v across a facet.

ufl.operators.le(left, right)
UFL operator: A boolean expression (left <= right) for use with conditional.

ufl.operators.ln(f)
UFL operator: Take the natural logarithm of f.

ufl.operators.lt(left, right)
UFL operator: A boolean expression (left < right) for use with conditional.

ufl.operators.max_value(x, y)
UFL operator: Take the maximum of x and y.

ufl.operators.min_value(x, y)
UFL operator: Take the minimum of x and y.
ufl.operators.nabla_div(f)
UFL operator: Take the divergence of \( f \).
This operator follows the div convention where
\[ \nabla \text{div}(v) = v[i].dx(i) \]
\[ \nabla \text{div}(T)[:,i] = T[:,i].dx(i) \]
for vector expressions \( v \), and arbitrary rank tensor expressions \( T \).
See also: \textit{div}()

ufl.operators.nabla_grad(f)
UFL operator: Take the gradient of \( f \).
This operator follows the grad convention where
\[ \nabla \text{grad}(s)[i] = s.dx(i) \]
\[ \nabla \text{grad}(v)[i,j] = v[j].dx(i) \]
\[ \nabla \text{grad}(T)[i, :] = T[:, i].dx(i) \]
for scalar expressions \( s \), vector expressions \( v \), and arbitrary rank tensor expressions \( T \).
See also: \textit{grad}()

ufl.operators.ne(left, right)
UFL operator: A boolean expression (left != right) for use with conditional.

ufl.operators.outer(*operands)
UFL operator: Take the outer product of two or more operands. The complex conjugate of the first argument is taken.

ufl.operators.perp(v)
UFL operator: Take the perp of \( v \), i.e. \((-v_1, +v_0)\).

ufl.operators.rank(f)
UFL operator: The rank of \( f \).

ufl.operators.real(f)
UFL operator: The real part of \( f \).

ufl.operators.rot(f)
UFL operator: Take the curl of \( f \).

ufl.operators.shape(f)
UFL operator: The shape of \( f \).

ufl.operators.sign(x)
UFL operator: Take the sign (+1 or -1) of \( x \).

ufl.operators.sin(f)
UFL operator: Take the sine of \( f \).

ufl.operators.sinh(f)
UFL operator: Take the hyperbolic sine of \( f \).

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 hyperbolic tangent 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().

1.3.34 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.

1.3.35 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=<class 'str'>)
### 1.3.36 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)**

### 1.3.37 ufl.referencevalue module

Representation of the reference value of a function.

```python
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
```

### 1.3.38 ufl.restriction module

Restriction operations.

```python
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)
        Evaluate expression at given coordinate with given values for terminals.

    side()

    ufl_free_indices
    ufl_index_dimensions
    ufl_shape
```

### 1.3.39 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.Directionalsobolevspace(orders)
Bases: ufl.sobolevspace.SobolevSpace

Symbolic representation of a Sobolev space with varying smoothness in different spatial directions.

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.

1.3.40 ufl.sorting module

This module contains a sorting rule for expr objects that is more robust w.r.t. argument numbering than using repr.

ufl.sorting.cmp_expr(a, b)
Replacement for cmp(a, b), removed in Python 3, for Expr objects.

ufl.sorting.sorted_expr(sequence)
Return a canonically sorted list of Expr objects in sequence.

ufl.sorting.sorted_expr_sum(seq)

1.3.41 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.

1.3.42 ufl.tensoralgebra module

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 = ()
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 = ()
    ufl_shape

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

1.3. ufl package
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

1.3.43 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)
        Evaluate expression at given coordinate with given values for terminals.
    indices()
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape
class ufl.tensors.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=())
        Evaluate expression at given coordinate with given values for terminals.
    ufl_free_indices
    ufl_index_dimensions
    ufl_shape

ufl.tensors.as_matrix(expressions, indices=None)
    UFL operator: As as_tensor(), but limited to rank 2 tensors.

ufl.tensors.as_scalar(expression)
    Given a scalar or tensor valued expression A, returns either of the tuples:

    \[(a, b) = (A, ())\]
    \[(a, b) = (A[indices], indices)\]

    such that a is always a scalar valued expression.

ufl.tensors.as_scalars(*expressions)
    Given multiple scalar or tensor valued expressions A, returns either of the tuples:
such that a is always a list of scalar valued expressions.

\[(a, b) = (A, ())\]
\[(a, b) = ([A[0][indices], ..., A[-1][indices]], indices)\]

such that a is always a list of scalar valued expressions.

**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 \(A[indices] = expressions\). 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.tensors.as_vector** *(expressions, index=None)*

UFL operator: As as_tensor(), but limited to rank 1 tensors.

**ufl.tensors.dyad**(d, *iota*)

TODO: Develop this concept, can e.g. write \(A[i,j]*\text{dyad}(j,i)\) for the transpose.

**ufl.tensors.from_numpy_to_lists**(expressions)

**ufl.tensors.numpy2nestedlists**(arr)

**ufl.tensors.relabel**(A, indexmap)

UFL operator: Relabel free indices of \(A\) with new indices, using the given mapping.

**ufl.tensors.unit_indexed_tensor**(shape, component)

**ufl.tensors.unit_list**(i, n)

**ufl.tensors.unit_list2**(i, j, n)

**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)

### 1.3.44 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**

**ufl_domains**

Return tuple of domains related to this terminal object.
ufl_free_indices
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

ufl_index_dimensions
Built-in immutable sequence.
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.
If the argument is a tuple, the return value is the same object.

ufl_shape

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:

```python
e = <...>
e = variable(e)
f = exp(e**2)
df = diff(f, e)
```

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

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

ufl_free_indices = ()
ufl_index_dimensions = ()
ufl_shape

1.3.45 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 the mathematical one.

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 an UFL expression tree is built from, type

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

• Various algorithms for working with UFL expression trees can be accessed by
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
  - vertex
  - interval
  - triangle
  - tetrahedron
  - quadrilateral
  - hexahedron

• Domains:
  - AbstractDomain
  - Mesh
  - MeshView
  - TensorProductMesh

• Sobolev spaces:
  - L2
  - H1
  - H2
  - HDiv
  - H Curl

• Elements:
  - FiniteElement
  - MixedElement
  - VectorElement
  - TensorElement
  - EnrichedElement
  - NodalEnrichedElement
  - RestrictedElement
  - TensorProductElement
  - HDivElement
  - H CurlElement
  - BrokenElement
  - FacetElement
  - InteriorElement
• Function spaces:
  - FunctionSpace
  - MixedFunctionSpace

• 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
  - CellDiameter
  - 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, nabla_grad
- div, nabla_div
- curl, rot
- Dx, Dn

• 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

• Complex operations:
- conj, real, imag
  conjugate is an alias for conj

• Discontinuous Galerkin operators:
- v('+'), v('-')
- jump
- avg
- 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, dX
  - ds_b, ds_t, ds_tb, 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: 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.

`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.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()  
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_facet_edges()  
The number of facet edges.

num_facets()  
The number of cell facets.

num_vertices()  
The number of cell vertices.

reconstruct(geometric_dimension=None)

class ufl.TensorProductCell(*cells, **kwargs)
Bases: ufl.cell.AbstractCell

cellname()  
Return the cellname of the cell.

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.

reconstruct(geometric_dimension=None)

sub_cells()  
Return list of cell factors.

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.
is_piecewise_linear_simplex_domain()

ufl_cargo()

Return carried object that will not be used by UFL.

ufl_cell()

ufl_coordinate_element()

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.

count()

evaluate(x, mapping, component, index_values)

Return the value of the coordinate.

is_cellwise_constant()

Return whether this expression is spatially constant over each cell.

name = 'x'

ufl_shape

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

class ufl.CellVolume(domain)

Bases: ufl.geometry.GeometricCellQuantity

UFL geometry representation: The volume of the cell.

name = 'volume'
class ufl.CellDiameter(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The diameter of the cell, i.e., maximal distance of two points in the cell.
    name = 'diameter'

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
        Return the number of coordinates defined (i.e. the geometric dimension of the domain).

class ufl.CellNormal(domain)
    Bases: ufl.geometry.GeometricCellQuantity
    UFL geometry representation: The upwards pointing normal vector of the current manifold cell.
    name = 'cell_normal'
**ufl_shape**

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

```python
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'
ufl_shape
```

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

```python
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'
```

```python
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'
ufl_shape
```

Return the number of coordinates defined (i.e. the geometric dimension of the domain).

```python
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.
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 \((\text{component=}\text{None})\)
Return whether the basis functions of this element is spatially constant over each cell.

mapping()
Not implemented.

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 \(c_0 \rightarrow c_1\) meaning that component \(c_0\) is represented by
component \(c_1\). A component is a tuple of one or more ints.

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 \((\text{family=}, \text{cell=}\text{None}, \text{degree=}\text{None}, \text{form_degree=}\text{None}, \text{quad_scheme=}\text{None}, \text{variant=}\text{None})\)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

The basic finite element class for all simple finite elements.

mapping()
Not implemented.

reconstruct \((\text{family=}\text{None}, \text{cell=}\text{None}, \text{degree=}\text{None}, \text{quad_scheme=}\text{None}, \text{variant=}\text{None})\)
Construct a new FiniteElement object with some properties replaced with new values.

shortstr()
Format as string for pretty printing.

sobolev_space()
Return the underlying Sobolev space.

variant()
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 reference_component index.

is_cellwise_constant (component=None)
    Return whether the basis functions of this element is spatially constant over each cell.

mapping ()
    Not implemented.

num_sub_elements ()
    Return number of sub elements.

reconstruct (**kwargs)

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 \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.VectorElement (family, cell=None, degree=None, 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.

reconstruct (**kwargs)

shortstr ()
    Format as string for pretty printing.

class ufl.TensorElement (family, cell=None, degree=None, 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()
Not implemented.

reconstruct (**kwargs)

shortstr()
Format as string for pretty printing.

symmetry()
Return the symmetry dict, which is a mapping \(c_0 \rightarrow c_1\) meaning that component \(c_0\) is represented by component \(c_1\). A component is a tuple of one or more ints.

class ufl.EnrichedElement(*elements)
Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

The vector sum of several finite element spaces:

\[
\text{EnrichedElement}(V,Q) = \{ v + q | v \in V, q \in Q \}.
\]

Dual basis is a concatenation of subelements dual bases; primal basis is a concatenation of subelements primal bases; resulting element is not nodal even when subelements are. Structured basis may be exploited in form compilers.

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

shortstr()
Format as string for pretty printing.

class ufl.NodalEnrichedElement(*elements)
Bases: ufl.finiteelement.enrichedelement.EnrichedElementBase

The vector sum of several finite element spaces:

\[
\text{EnrichedElement}(V,Q) = \{ v + q | v \in V, q \in Q \}.
\]

Primal basis is reorthogonalized to dual basis which is a concatenation of subelements dual bases; resulting element is nodal.

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

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.

is_cellwise_constant()
Return whether the basis functions of this element is spatially constant over each cell.

mapping()
Not implemented.

num_restricted_sub_elements()
Return number of restricted sub elements.
num_sub_elements()
    Return number of sub elements.
reconstruct(**kwargs)
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 \( c_0 \rightarrow c_1 \) meaning that component \( c_0 \) is represented by component \( c_1 \). A component is a tuple of one or more ints.

class ufl.TensorProductElement(*elements, **kwargs)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
The tensor product of \( d \) element spaces:
\[
V = V_1 \otimes V_2 \otimes \ldots \otimes V_d
\]
Given bases \( \{ \phi_{j_i} \} \) of the spaces \( V_i \) for \( i = 1, \ldots, d \), \( \{ \phi_{j_1} \otimes \phi_{j_2} \otimes \ldots \otimes \phi_{j_d} \} \) forms a basis for \( V \).
mapping()
    Not implemented.
num_sub_elements()
    Return number of subelements.
reconstruct(cell=None)
shortstr()
    Short pretty-print.
sobolev_space()
    Return the underlying Sobolev space of the TensorProductElement.
sub_elements()
    Return subelements (factors).

class ufl.HDivElement(element)
Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase
A div-conforming version of an outer product element, assuming this makes mathematical sense.
mapping()
    Not implemented.
reconstruct(**kwargs)
shortstr()
    Format as string for pretty printing.
sobolev_space()
    Return the underlying Sobolev space.
class ufl.HCurlElement(element)
   Bases: ufl.finiteelement.finiteelementbase.FiniteElementBase

   A curl-conforming version of an outer product element, assuming this makes mathematical sense.
   
   mapping()
      Not implemented.

   reconstruct(**kwargs)

   shortstr()
      Format as string for pretty printing.

   sobolev_space()
      Return the underlying Sobolev space.

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

   The discontinuous version of an existing Finite Element space.
   
   mapping()
      Not implemented.

   reconstruct(**kwargs)

   shortstr()
      Format as string for pretty printing.

ufl.FacetElement(element)
   Constructs the restriction of a finite element to the facets of the cell.

ufl.InteriorElement(element)
   Constructs the restriction of a finite element to the interior of the cell.

ufl.register_element(family, short_name, value_rank, sobolev_space, mapping, degree_range, cell_names)
   Register new finite element family.

ufl.show_elements()
   Shows all registered elements.

class ufl.FunctionSpace(domain, element)
   Bases: ufl.functionspace.AbstractFunctionSpace

   ufl_domain()
      Return ufl domain.

   ufl_domains()
      Return ufl domains.

   ufl_element()
      Return ufl element.

   ufl_sub_spaces()
      Return ufl sub spaces.

class ufl.MixedFunctionSpace(*args)
   Bases: ufl.functionspace.AbstractFunctionSpace

   num_sub_spaces()

   ufl_domain()
      Return ufl domain.
ufl_domains()
   Return ufl domains.

ufl_element()

ufl_elements()
   Return ufl elements.

ufl_sub_space(i)
   Return i-th ufl sub space.

ufl_sub_spaces()
   Return 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.

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

   number()
      Return the Argument number.

part()

ufl_domain()
   Deprecated, please use .ufl_function_space().ufl_domain() instead.

ufl_domains()
   Deprecated, please use .ufl_function_space().ufl_domains() instead.

ufl_element()
   Deprecated, please use .ufl_function_space().ufl_element() instead.

ufl_function_space()
   Get the function space of this Argument.

ufl_shape
   Return the associated 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 correspond-
   ing to the subelements.

ufl.TestFunctions (function_space)
   UFL value: Create a TestFunction in a mixed space, and return a tuple with the function components corre-
   sponding to the subelements.

ufl.TrialFunctions (function_space)
   UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corre-
   sponding 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()

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
   Return the associated UFL shape.

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.

class ufl.Constant(domain, shape=(), count=None)
   Bases: ufl.core.terminal.Terminal

count()

is_cellwise_constant()

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

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

ufl_shape

ufl.VectorConstant(domain, count=None)

ufl.TensorConstant(domain, count=None)

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)
      Evaluates the permutation symbol.

ufl_shape

class ufl.Identity(dim)
   Bases: ufl.constantvalue.ConstantValue

   UFL literal type: Representation of an identity matrix.

   evaluate(x, mapping, component, index_values)
      Evaluates the identity matrix on the given components.
**ufl_shape**

```python
ufl.zero(*shape)
```

UFL literal constant: Return a zero tensor with the given shape.

```python
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.

```python
count()
```

**ufl.indices**(n)

UFL value: Return a tuple of n new Index objects.

```python
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.

```python
ufl.as_vector(expressions, index=None)
```

UFL operator: As as_tensor(), but limited to rank 1 tensors.

```python
ufl.as_matrix(expressions, indices=None)
```

UFL operator: As as_tensor(), but limited to rank 2 tensors.

```python
ufl.relabel(A, indexmap)
```

UFL operator: Relabel free indices of $A$ with new indices, using the given mapping.

```python
ufl.unit_vector(i, d)
```

UFL value: A constant unit vector in direction $i$ with dimension $d$.

```python
ufl.unit_vectors(d)
```

UFL value: A tuple of constant unit vectors in all directions with dimension $d$.

```python
ufl.unit_matrix(i, j, d)
```

UFL value: A constant unit matrix in direction $i$, $j$ with dimension $d$.

```python
ufl.unit_matrices(d)
```

UFL value: A tuple of constant unit matrices in all directions with dimension $d$.

```python
ufl.rank(f)
```

UFL operator: The rank of $f$.

```python
ufl.shape(f)
```

UFL operator: The shape of $f$.

```python
ufl.conj(f)
```

UFL operator: The complex conjugate of $f$.

```python
ufl.real(f)
```

UFL operator: The real part of $f$.
**UFL operator**: The imaginary part of $f$

**UFL operator**: Take the outer product of two or more operands. The complex conjugate of the first argument is taken.

**UFL operator**: Take the inner product of $a$ and $b$. The complex conjugate of the second argument is taken.

**UFL operator**: Take the dot product of $a$ and $b$. The complex conjugate of the second argument is taken.

**UFL operator**: Take the cross product of $a$ and $b$.

**UFL operator**: Take the perp of $v$, i.e. $(-v_1, +v_0)$.

**UFL operator**: Take the determinant of $A$.

**UFL operator**: Take the inverse of $A$.

**UFL operator**: Take the cofactor of $A$.

**UFL operator**: Take the transposed of tensor $A$.

**UFL operator**: Take the trace of $A$.

**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 operator**: Take the diagonal part of rank 2 tensor $A$ and return as a vector. See also `diag`.

**UFL operator**: Take the deviatoric part of $A$.

**UFL operator**: Take the skew symmetric part of $A$.

**UFL operator**: Take the symmetric part of $A$.

**UFL operator**: Take the square root of $f$.

**UFL operator**: Take the exponential of $f$.

**UFL operator**: Take the natural logarithm of $f$.

**UFL operator**: Take the error function of $f$. 

---

**1.3. ufl package**
ufl.cos(f)
    UFL operator: Take the cosine of f.

ufl.sin(f)
    UFL operator: Take the sine of f.

ufl.tan(f)
    UFL operator: Take the tangent of f.

ufl.acos(f)
    UFL operator: Take the inverse cosine of f.

ufl.asin(f)
    UFL operator: Take the inverse sine of f.

ufl.atan(f)
    UFL operator: Take the inverse tangent of f.

ufl.atan_2(f1,f2)
    UFL operator: Take the inverse tangent with two the arguments f1 and f2.

ufl.cosh(f)
    UFL operator: Take the hyperbolic cosine of f.

ufl.sinh(f)
    UFL operator: Take the hyperbolic sine of f.

ufl.tanh(f)
    UFL operator: Take the hyperbolic tangent 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 expression (left == right) for use with conditional.

ufl.ne(left, right)
    UFL operator: A boolean expression (left != right) for use with conditional.

ufl.le(left, right)
    UFL operator: A boolean expression (left <= right) for use with conditional.

ufl.ge(left, right)
    UFL operator: A boolean expression (left >= right) for use with conditional.

ufl.lt(left, right)
    UFL operator: A boolean expression (left < right) for use with conditional.

ufl.gt(left, right)
    UFL operator: A boolean expression (left > right) for use with conditional.

ufl.And(left, right)
    UFL operator: A boolean expression (left and right) for use with conditional.
### UFL operators

- **UFL operators** are used for various mathematical operations in the Unified Form Language (UFL).

#### Or (left, right)
- UFL operator: A boolean expression (left or right) for use with `conditional`.

#### Not (condition)
- UFL operator: A boolean expression (not condition) for use with `conditional`.

#### 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.

#### sign (x)
- UFL operator: Take the sign (+1 or -1) of `x`.

#### max_value (x, y)
- UFL operator: Take the maximum of `x` and `y`.

#### min_value (x, y)
- UFL operator: Take the minimum of `x` and `y`.

#### Max (x, y)
- UFL operator: Take the maximum of `x` and `y`.

#### Min (x, y)
- UFL operator: Take the minimum of `x` and `y`.

#### variable (e)
- UFL operator: Define a variable representing the given expression, see also `diff()`.

#### 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.

#### Dx (f, *i)
- UFL operator: Take the partial derivative of `f` with respect to spatial variable number `i`. Equivalent to `f.dx(*i)`.

#### grad (f)
- UFL operator: Take the gradient of `f`.
  - This operator follows the grad convention where
    - `grad(s)[i] = s.dx(i)`
    - `grad(v)[i,j] = v[i].dx(j)`
    - `grad(T)[:,:i] = T[::].dx(i)`
  - for scalar expressions `s`, vector expressions `v`, and arbitrary rank tensor expressions `T`.
  - See also: `nabla_grad()`

#### div (f)
- UFL operator: Take the divergence of `f`.
  - This operator follows the div convention where
    - `div(v) = v[i].dx(i)`
    - `div(T)[i] = T[::].dx(i)`
  - for vector expressions `v`, and arbitrary rank tensor expressions `T`.
  - See also: `nabla_div()`

---

**1.3. ufl package**
ufl.curl(f)
    UFL operator: Take the curl of $f$.

ufl.rot(f)
    UFL operator: Take the curl of $f$.

ufl.nabla_grad(f)
    UFL operator: Take the gradient of $f$.
    This operator follows the grad convention where
    \[
    \nabla_{\text{grad}}(s)[i] = s.dx(i) \\
    \nabla_{\text{grad}}(v)[i,j] = v[j].dx(i) \\
    \nabla_{\text{grad}}(T)[i,:] = T[:,i].dx(i)
    \]
    for scalar expressions $s$, vector expressions $v$, and arbitrary rank tensor expressions $T$.
    See also: grad()

ufl.nabla_div(f)
    UFL operator: Take the divergence of $f$.
    This operator follows the div convention where
    \[
    \nabla_{\text{div}}(v) = v[i].dx(i) \\
    \nabla_{\text{div}}(T)[i,:] = T[i,:].dx(i)
    \]
    for vector expressions $v$, and arbitrary rank tensor expressions $T$.
    See also: div()

ufl.Dn(f)
    UFL operator: Take the directional derivative of $f$ in the facet normal direction, $D_n(f) := \text{dot}(\text{grad}(f), n)$.

ufl.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.jump(v, n=None)
    UFL operator: Take the jump of $v$ across a facet.

ufl.avg(v)
    UFL operator: Take the average of $v$ across a facet.

ufl.cell_avg(f)
    UFL operator: Take the average of $v$ over a cell.

ufl.facet_avg(f)
    UFL operator: Take the average of $v$ over a facet.

ufl.elem_mult(A, B)
    UFL operator: Take the elementwise multiplication of tensors $A$ and $B$ with the same shape.

ufl.elem_div(A, B)
    UFL operator: Take the elementwise division of tensors $A$ and $B$ with the same shape.

ufl.elem_pow(A, B)
    UFL operator: Take the elementwise power of tensors $A$ and $B$ with the same shape.
**ufl.elem_op(op, *args)**
UFL operator: Take the elementwise application of operator `op` on scalar values from one or more tensor arguments.

**class ufl.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.

**coefficient_numbering()**
Return a contiguous numbering of coefficients in a mapping `{coefficient: number}`.

**coefficients()**
Return all Coefficient objects found in form.

**constants()**

**domain_numbering()**
Return a contiguous numbering of domains in a mapping `{domain: number}`.

**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_domain(domain)**
Return a sequence of all integrals with a particular integration domain.

**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.

```python
class ufl.Integral(integrand, integral_type, domain, subdomain_id, metadata, subdomain_data)
```

Bases: `object`

An integral over a single domain.

```python
integral_type()
```

Return the domain type of this integral.

```python
integrand()
```

Return the integrand expression, which is an `Expr` instance.

```python
metadata()
```

Return the compiler metadata this integral has been annotated with.

```python
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})
```

```python
subdomain_data()
```

Return the domain data of this integral.

```python
subdomain_id()
```

Return the subdomain id of this integral.

```python
ufl_domain()
```

Return the integration domain of this integral.

```python
class ufl.Measure(integral_type, domain=None, subdomain_id='everywhere', metadata=None, subdomain_data=None)
```

Bases: `object`

```python
integral_type()
```

Return the domain type.

Valid domain types are "cell", "exterior_facet", "interior_facet", etc.

```python
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.

```python
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})
```

```python
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.

```python
subdomain_id()
```

Return the domain id of this measure (integer).

```python
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, sometimes occurring in pydolfin, e.g. assemble(1*dx, mesh=mesh).

ufl.derivative(form, coefficient=None, 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 Coefficient s 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:
    \[
    \begin{align*}
    a &= u\cdot v\,dx + f\cdot v\,dx \\
    L &= \text{rhs}(a) \rightarrow -f\cdot v\,dx
    \end{align*}
    \]

ufl.lhs(form)
    UFL form operator: Given a combined bilinear and linear form, extract the left hand side (bilinear form part).
    Example:
    \[
    \begin{align*}
    a &= u\cdot v\,dx + f\cdot v\,dx \\
    a &= \text{lhs}(a) \rightarrow u\cdot v\,dx
    \end{align*}
    \]

ufl.extract_blocks(form, i=None, j=None)
    UFL form operator: Given a linear or bilinear form on a mixed space, extract the block corresponding to the
    indices ix, iy.
    Example:
a = inner(grad(u), grad(v))*dx + div(u)*q*dx + div(v)*p*dx
evaluate_blocks(a, 0, 0) -> inner(grad(u), grad(v))*dx
evaluate_blocks(a) -> [inner(grad(u), grad(v))*dx, div(v)*p*dx, div(u)*q*dx, 0]

**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, and taking the complex conjugate of the result.

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.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

\[ 0 = \frac{d}{dv}(Ax - b) = \frac{dA}{dv}x + A \frac{dx}{dv} - \frac{db}{dv}, \]

\[ A \frac{dx}{dv} = \frac{db}{dv} - \frac{dA}{dv}x, \]

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

```python
v = variable(v_expression)
L = IL(v)*dx
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{du}{dv}` and `\frac{dA}{dv}` from the forms

```python
da = diff(a, v)
dL = diff(L, v)
```

and the action of `da` on `u` by

```python
da_u = action(da, u)
```

In total, we can build the right hand side of the system to compute `\frac{du}{dv}` with the single line

```python
dL = diff(L, v) - action(diff(a, v), u)
```

or, using this function,

```python
dL = sensitivity_rhs(a, u, L, v)
```
1.4 Release notes

1.4.1 Changes in the next release

Summary of changes

Note: Developers should use this page to track and list changes during development. At the time of release, this page should be published (and renamed) to list the most important changes in the new release.

Detailed changes

Note: At the time of release, make a verbatim copy of the ChangeLog here (and remove this note).

1.4.2 Changes in version 2019.1.0

Summary of changes

• Add support for complex valued elements
• Remove LaTeX support (not functional)
• Remove scripts

Detailed changes

• Add support for complex valued elements; complex mode is chosen by `compute_form_data(form, complex_mode=True)` typically by a form compiler; otherwise UFL language is agnostic to the choice of real/complex domain

1.4.3 Changes in version 2018.1.0

UFL 2018.1.0 was released on 2018-06-14.

Summary of changes

• Remove python2 support.

1.4.4 Changes in version 2017.2.0

UFL 2017.2.0 was released on 2017-12-05.
Summary of changes

- Add `CellDiameter` expression giving diameter of a cell, i.e., maximal distance between any two points of the cell. Implemented for all simplices and quads/hexes.
- Make `(Min|Max)(Cell|Facet)EdgeLength` working for quads/hexes

Detailed changes

- Add geometric quantity `CellDiameter` defined as a set diameter of the cell, i.e., maximal distance between any two points of the cell; implemented on simplices and quads/hexes
- Rename internally used reference quantities `(Cell|Facet)EdgeVectors` to `Reference(Cell|Facet)EdgeVectors`
- Add internally used quantities `CellVertices`, `(Cell|Facet)EdgeVectors` which are physical-coordinates-valued; will be useful for further geometry lowering implementations for quads/hexes
- Implement geometry lowering of `(Min|Max)(Cell|Facet)EdgeLength` for quads and hexes

1.4.5 Changes in version 2017.1.0.post1

UFL 2017.1.0.post1 was released on 2017-09-12.

Summary of changes

- Change PyPI package name to fenics-ufl.

1.4.6 Changes in version 2017.1.0

UFL 2017.1.0 was released on 2017-05-09.

Summary of changes

- Add the `DirectionalSobolevSpace` subclass of `SobolevSpace`. This allows one to use spaces where elements have varying continuity in different spatial directions.
- Add `sobolev_space` methods for `HDiv` and `HCurl` finite elements.
- Add `sobolev_space` methods for `TensorProductElement` and `EnrichedElement`. The smallest shared Sobolev space will be returned for enriched elements. For the tensor product elements, a `DirectionalSobolevSpace` is returned depending on the order of the spaces associated with the component elements.

1.4.7 Changes in version 2016.2.0

UFL 2016.2.0 was released on 2016-11-30.
Summary of changes

- 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
- OuterProductCell and OuterProductElement are merged into TensorProductCell and TensorProductElement respectively
- Better degree estimation for quadrilaterals
- Expansion rules for Q, DQ, RTCE, RTCF, NCE and NCF on tensor product cells
- Add discontinuous Taylor elements
- Add support for the mapping double covariant Piola in uflacs
- Add support for the mapping double contravariant Piola in uflacs
- Support for tensor-valued subelements in uflacs fixed
- Replacing Discontinuous Lagrange Trace with HDiv Trace and removing TraceElement
- Assigning Discontinuous Lagrange Trace and DGT as aliases for HDiv Trace

Detailed changes

- Add call operator syntax to Form to replace arguments and coefficients. This makes it easier to e.g. express the norm defined by a bilinear form as a functional. Example usage:

```
# Equivalent to replace(a, {u: f, v: f})
M = a(f, f)
# Equivalent to replace(a, {f:1})
c = a(coefficients={f:1})
```

- Add call operator syntax to Form to replace arguments and coefficients::
  
  \[ a(f, g) == replace(a, \{u: f, v: g\}) \]
  
  \[ a(coefficients=\{f:1\}) == replace(a, \{f:1\}) \]

- Add @ operator to Form: form @ f == action(form, f) (python 3.5+ only)
- Reduce noise in Mesh str such that print(form) gets more short and readable
- Fix repeated split(function) for arbitrary nested elements
- **EnrichedElement: Remove */# warning** In the distant past, A + B => MixedElement([A, B]). The change that A + B => EnrichedElement([A, B]) was made in d522c74 (22 March 2010). A warning was introduced in fbc5ff (26 March 2010) that the meaning of “+” had changed, and that users wanting a MixedElement should use “*” instead. People have, presumably, been seeing this warning for 6 1/2 years by now, so it’s probably safe to remove.
- Rework TensorProductElement implementation, replaces OuterProductElement
- Rework TensorProductCell implementation, replaces OuterProductCell
- Remove OuterProductVectorElement and OuterProductTensorElement
- Add FacetElement and InteriorElement
• Add Hellan-Herrmann-Johnson element
• Add support for double covariant and contravariant mappings in mixed elements
• Support discontinuous Taylor elements on all simplices
• Some more performance improvements
• Minor bugfixes
• Improve Python 3 support
• More permissive in integer types accepted some places
• Make ufl pass almost all flake8 tests
• Add bitbucket pipelines testing
• Improve documentation

1.4.8 Changes in version 2016.1.0

UFL 2016.1.0 was released on 2016-06-23.

• Add operator $A^{(i,j)} := as_tensor(A, (i,j))$
• Updates to old manual for publishing on fenics-ufl.readthedocs.org
• Bugfix for ufl files with utf-8 encoding
• **Bugfix in conditional derivatives to avoid inf/nan values in generated code.** This bugfix may break ffc if uflacs is not used, to get around that the old workaround in ufl can be enabled by setting ufl.algorithms.apply_derivatives.CONDITIONAL_WORKAROUND = True at the top of your program.
• Allow sum([expressions]) where expressions are nonscalar by defining expr+0==expr
• Allow form=0; form -= other;
• 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
• Add new class Mesh to replace Domain
• Add new class FunctionSpace(mesh, element)
• Make FiniteElement classes take Cell, not Domain.
• Large reworking of symbolic geometry pipeline
• Implement symbolic Piola mappings

1.4.9 Changes in version 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
• 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

[FIXME: These links don’t belong here, should go under API reference somehow.]

• genindex
• modindex
ufl.indexsum, 144
ufl.integral, 145
ufl.log, 145
ufl.mathfunctions, 147
ufl.measure, 148
ufl.objects, 149
ufl.operators, 149
ufl.permutation, 154
ufl.precedence, 154
ufl.protocols, 155
ufl.referencevalue, 155
ufl.restriction, 155
ufl.sobolevspace, 155
ufl.sorting, 156
ufl.split_functions, 156
ufl.tensoralgebra, 156
ufl.tensors, 158
ufl.variable, 159
Index

A
Abs (class in ufl.algebra), 84
Abs (class in ufl.classes), 103
abs() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
abs() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
abs() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
AbstractCell (class in ufl), 164
AbstractCell (class in ufl.cell), 87
AbstractCell (class in ufl.classes), 112
AbstractDomain (class in ufl), 165
AbstractDomain (class in ufl.classes), 118
AbstractDomain (class in ufl.domain), 129
AbstractDomain (class in ufl.functionspace), 135
accumulate_integrands_with_same_metadata() (in module ufl.algorithms.domain_analysis), 61
Acos (class in ufl.classes), 110
Acos (class in ufl.mathfunctions), 147
acos() (in module ufl), 178
acos() (in module ufl.operators), 150
acos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
action() (in module ufl), 183
action() (in module ufl.formoperators), 133
add_indent() (ufl.log.Logger method), 145
add_logfile() (ufl.log.Logger method), 145
adj_expr() (in module ufl.compound_expressions), 123
adj_expr_2x2() (in module ufl.compound_expressions), 123
adj_expr_3x3() (in module ufl.compound_expressions), 123
adj_expr_4x4() (in module ufl.compound_expressions), 123
adjoint() (in module ufl), 184
adjoint() (in module ufl.formoperators), 133
affine_mesh() (in module ufl.domain), 130
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 method), 74
always_reconstruct() (ufl.algorithms.transformer.Transformer method), 72
analyse_key() (in module ufl.exproperators), 132
And() (in module ufl), 178
And() (in module ufl.operators), 149
AndCondition (class in ufl.classes), 109
AndCondition (class in ufl.conditional), 124
apply_algebra_lowering() (in module ufl.algorithms.apply_algebra_lowering), 48
apply_coordinate_derivatives() (in module ufl.algorithms.apply_derivatives), 53
apply_default_restrictions() (in module ufl.algorithms.apply_restrictions), 56
apply_derivatives() (in module ufl.algorithms.apply_derivatives), 53
apply_function_pullbacks() (in module ufl.algorithms.apply_function_pullbacks), 53
apply_geometric_lowering() (in module ufl.algorithms.apply_geometric_lowering), 54
apply_integral_scaling() (in module ufl.algorithms.apply_integral_scaling), 54
apply_restrictions() (in module ufl.algorithms.apply_restrictions), 56
apply_single_function_pullbacks() (in module ufl.algorithms.apply_function_pullbacks), 53
apply_transformer() (in module ufl.algorithms), 74
apply_transformer() (in module ufl.algorithms.transformer), 73
Argument (class in ufl), 174
Argument (class in ufl.argument), 85
check_form_arity() (in module ufl.algorithms.check_arities), 59
check_integrand_arity() (in module ufl.algorithms.check_arities), 59
check_restrictions() (in module ufl.algorithms.check_restrictions), 59
CheckComparisons (class in ufl.algorithms.comparison_checker), 59
Circumradius (class in ufl), 167
Circumradius (class in ufl.classes), 96
Circumradius (class in ufl.geometry), 139
circumradius() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
cmp_expr() (in module ufl.sorting), 156
codeterminant_expr_nxn() (in module ufl.compound_expressions), 123
Coefficient (class in ufl), 174
Coefficient (class in ufl.classes), 101
Coefficient (class in ufl.coefficient), 122
coefficient() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleDispatcher method), 48
coefficient() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
coefficient() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
coefficient() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 57
coefficient() (ufl.algorithms.replace.Replacer method), 57
coefficient_numbering() (ufl.classes.Form method), 121
coefficient_numbering() (ufl.Form method), 181
CoefficientDerivative (class in ufl.classes), 106
CoefficientDerivative (class in ufl.differentiation), 127
Coefficients() (in module ufl), 175
Coefficients() (in module ufl.coefficient), 123
coefficients() (ufl.classes.Form method), 121
coefficients() (ufl.Form method), 181
coefficients() (ufl.form.Form method), 132
cofac() (in module ufl), 177
cofac() (in module ufl.operators), 150
Cofactor (class in ufl.classes), 105
Cofactor (class in ufl.tensoralgebra), 156
cofactor() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cofactor() (ufl.algorithms.apply_algebra_lowering.LowerCompoundTensorOperator method), 47
cofactor() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
cofactor_expr() (in module ufl.compound_expressions), 123
cofactor_expr_2x2() (in module ufl.compound_expressions), 123
cofactor_expr_4x4() (in module ufl.compound_expressions), 123
cmp_form() (in module ufl.algorithms), 75
compute_energy_norm() (in module ufl.algorithms.formtransformations), 69
compute_energy_norm() (in module ufl.algorithms.formtransformations), 69
compute_expression_hashdata() (in module ufl.algorithms.signature), 71
compute_expression_signature() (in module ufl.algorithms.signature), 71
compound_derivative() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
compound_tensor_operator() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
ComponentTensor (class in ufl.classes), 100
ComponentTensor (class in ufl.classes), 104
ComponentTensor (class in ufl.tensoralgebra), 158
compute_energy_norm() (in module ufl.algorithms.formtransformations), 69
compute_energy_norm() (in module ufl.algorithms.formtransformations), 69
compute_expression_hashdata() (in module ufl.algorithms.signature), 71
compute_expression_signature() (in module ufl.algorithms.signature), 71
Index
CopyTransformer (class in ufl.algorithms.transformer), 71
Cos (class in ufl.classes), 110
Cos (class in ufl.mathfunctions), 147
cos() (in module ufl), 177
cos() (in module ufl.operators), 150
cos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
Cosh (class in ufl.classes), 110
Cosh (class in ufl.mathfunctions), 147
cosh() (in module ufl), 178
cosh() (in module ufl.operators), 151
cosh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
count() (ufl.classes.Coefficient method), 101
count() (ufl.classes.Constant method), 101
count() (ufl.classes.Index method), 119
count() (ufl.classes.Label method), 101
count() (ufl.classes.SpatialCoordinate method), 90
count() (ufl.Class method), 174
count() (ufl.coefficient.Coefficient method), 122
count() (ufl.Constant method), 175
count() (ufl.constant.Constant method), 125
count() (ufl.geometry.SpatialCoordinate method), 143
count() (ufl.Index method), 176
count() (ufl.SpatialCoordinate method), 166
count() (ufl.variable.Label method), 159
create_nested_lists() (in module ufl.algorithms.apply_function_pullbacks), 53
create_slice_indices() (in module ufl.index_combination_utils), 144
Cross (class in ufl.classes), 104
Cross (class in ufl.tensoralgebra), 156
cross() (in module ufl), 177
cross() (in module ufl.operators), 151
cross() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cross() (ufl.geometry.SpatialCoordinate method), 143
cross() (ufl.geometry.SpatialCoordinate method), 143
cross() (ufl.tensoralgebra.Determinant method), 156
cross() (ufl.tensoralgebra.Determinant method), 156
cross() (ufl.tensoralgebra.Deviatoric method), 157
cross() (ufl.tensoralgebra.Deviatoric method), 157
cross_expr() (in module ufl.compound_expressions), 123
cross_expr() (in module ufl.compound_expressions), 123
curl() (in module ufl), 179
curl() (in module ufl.operators), 151
curl() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
curl() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
curl() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
default_domain() (in module ufl.domain), 130
test() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
ufl.compound_expressions), 123
diag() (in module ufl), 177
diag() (in module ufl.operators), 151
diag_vector() (in module ufl), 177
diag_vector() (in module ufl.operators), 151
dicts_lt() (in module ufl.algorithms.domain_analysis), 62
diff() (in module ufl), 179
diff() (in module ufl.operators), 151
dimension() (ufl.classes.IndexSum method), 105
dimension() (ufl.indexsum.IndexSum method), 144
DirectionalSobolevSpace (class in ufl.sobolevspace), 155
Div (class in ufl.classes), 107
Div (class in ufl.differentiation), 128
div() (in module ufl), 179
div() (in module ufl.operators), 151
div() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
div() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
Division (class in ufl.algebra), 84
Division (class in ufl.classes), 103
division() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleSet method), 50
division() (ufl.algorithms.check_arities.ArityChecker method), 58
division() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
division() (ufl.algorithms.expand_indices.IndexExpander method), 66
division() (ufl.algorithms.formtransformations.PartExtractor method), 68
Dn() (in module ufl), 180
Dn() (in module ufl.operators), 149
do_comparison_check() (in module ufl.algorithms.comparison_checker), 60
domain (ufl.algorithms.domain_analysis.IntegralData attribute), 61
domain_numbering() (ufl.classes.Form method), 121
domain_numbering() (ufl.Form method), 181
domain_numbering() (ufl.form.Form method), 132
Dot (class in ufl.classes), 104
Dot (class in ufl.tensoralgebra), 157
dot() (in module ufl), 177
dot() (in module ufl.operators), 151
dot() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
dot() (ufl.algorithms.check_arities.ArityChecker method), 58
dot() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
dot() (ufl.algorithms.formtransformations.PartExtractor method), 68
Dt() (in module ufl.operators), 149
Dx() (in module ufl), 179
Dx() (in module ufl.operators), 150
dx() (ufl.classes.Expr method), 89
dyad() (in module ufl.tensors), 159

E
elem_div() (in module ufl), 180
elem_div() (in module ufl.operators), 151
elem_mult() (in module ufl), 180
elem_mult() (in module ufl.operators), 151
elem_op() (in module ufl), 180
elem_op() (in module ufl.operators), 151
elem_pow() (in module ufl), 180
elem_pow() (in module ufl.operators), 151
empty() (ufl.classes.Form method), 121
empty() (ufl.form.Form method), 132
enabled_coefficients (ufl.algorithms.domain_analysis[IntegralData attribute), 61
end() (ufl.log.Logger method), 146
ergy_norm() (in module ufl), 183
energy_form() (in module ufl.formoperators), 134
EnrichedElement (class in ufl), 171
EnrichedElement (class in ufl.classes), 115
EnrichedElement (class in ufl.finiteelement.enrichedelement), 78
EnrichedElementBase (class in ufl.finiteelement.enrichedelement), 78
EQ (class in ufl.classes), 108
EQ (class in ufl.conditional), 124
eq() (in module ufl), 178
eq() (in module ufl.operators), 151
equals() (ufl.classes.Form method), 121
equals() (ufl.Form method), 181
equals() (ufl.form.Form method), 132
Equation (class in ufl.classes), 122
Equation (class in ufl.equation), 131
Erf (class in ufl.classes), 111
Erf (class in ufl.mathfunctions), 147
erf() (in module ufl), 177
erf() (in module ufl.operators), 151
erf() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
error() (ufl.log.Logger method), 146
estimate_total_polynomial_degree() (in module ufl.algorithms), 73
estimate_total_polynomial_degree() (in module ufl.algorithms.estimate_degrees), 65
evaluate() (ufl.algebra.Abs method), 84
evaluate() (ufl.algebra.Conj method), 84
evaluate() (ufl.algebra.Division method), 84
evaluate() (ufl.algebra.Imag method), 84
evaluate() (ufl.algebra.Power method), 85
evaluate() (ufl.algebra.Product method), 85
evaluate() (ufl.algebra.Real method), 85
evaluate() (ufl.algebra.Sum method), 85
evaluate() (ufl.averaging.CellAvg method), 86
evaluate() (ufl.averaging.FacetAvg method), 87
evaluate() (ufl.classes.Abs method), 103
evaluate() (ufl.classes.AndCondition method), 109
evaluate() (ufl.classes.Atan2 method), 110
evaluate() (ufl.classes.BesselFunction method), 111
evaluate() (ufl.classes.CellAvg method), 111
evaluate() (ufl.classes.ComponentTensor method), 100
evaluate() (ufl.classes.Conditional method), 109
evaluate() (ufl.classes.Conj method), 103
evaluate() (ufl.classes.Division method), 103
evaluate() (ufl.classes.EQ method), 108
evaluate() (ufl.classes.Erf method), 111
evaluate() (ufl.classes.Expr method), 89
evaluate() (ufl.classes.FacetAvg method), 111
evaluate() (ufl.classes.Grad method), 107
evaluate() (ufl.classes.GT method), 109
evaluate() (ufl.classes.Identity method), 100
evaluate() (ufl.classes.Imag method), 104
evaluate() (ufl.classes.Indexed method), 100
evaluate() (ufl.classes.IndexSum method), 105
evaluate() (ufl.classes.LE method), 108
evaluate() (ufl.classes.ListTensor method), 100
evaluate() (ufl.classes.Ln method), 110
evaluate() (ufl.classes.LT method), 109
evaluate() (ufl.classes.MathFunction method), 110
evaluate() (ufl.classes.MaxValue method), 109
evaluate() (ufl.classes.MinValue method), 109
evaluate() (ufl.classes.MultiIndex method), 98
evaluate() (ufl.classes.NotCondition method), 109
evaluate() (ufl.classes.OrCondition method), 109
evaluate() (ufl.classes.PermutationSymbol method), 100
evaluate() (ufl.classes.Power method), 103
evaluate() (ufl.classes.Product method), 102
evaluate() (ufl.classes.Real method), 103
evaluate() (ufl.classes.ReferenceGrad method), 107
evaluate() (ufl.classes.ReferenceValue method), 112
evaluate() (ufl.classes.Restricted method), 106
evaluate() (ufl.classes.ScalarValue method), 99
evaluate() (ufl.classes.SpatialCoordinate method), 90
evaluate() (ufl.classes.Sum method), 102
evaluate() (ufl.classes.Terminal method), 90
evaluate() (ufl.classes.Variable method), 102
evaluate() (ufl.classes.Zero method), 98
evaluate() (ufl.conditional.AndCondition method), 124
evaluate() (ufl.conditional.Conditional method), 124
evaluate() (ufl.conditional.EQ method), 124
evaluate() (ufl.conditional.GE method), 124
evaluate() (ufl.conditional.GT method), 124
evaluate() (ufl.conditional.LE method), 124
evaluate() (ufl.conditional.LT method), 124
evaluate() (ufl.conditional.MaxValue method), 125
evaluate() (ufl.conditional.MinValue method), 125
evaluate() (ufl.conditional.NE method), 125
evaluate() (ufl.conditional.NotCondition method), 125
evaluate() (ufl.conditional.OrCondition method), 125
evaluate() (ufl.constantvalue.Identity method), 126
evaluate() (ufl.constantvalue.PermutationSymbol method), 126
evaluate() (ufl.constantvalue.ScalarValue method), 127
evaluate() (ufl.constantvalue.Zero method), 127
evaluate() (ufl.differentiation.Grad method), 128
evaluate() (ufl.differentiation.ReferenceGrad method), 129
evaluate() (ufl.geometry.SpatialCoordinate method), 143
evaluate() (ufl.Identity method), 175
evaluate() (ufl.indexed.Indexed method), 144
evaluate() (ufl.indexsum.IndexSum method), 144
evaluate() (ufl.mathfunctions.Atan2 method), 147
evaluate() (ufl.mathfunctions.BesselFunction method), 147
evaluate() (ufl.mathfunctions.Erf method), 147
evaluate() (ufl.mathfunctions.Ln method), 148
evaluate() (ufl.mathfunctions.MathFunction method), 148
evaluate() (ufl.mathfunctions.PermutationSymbol method), 175
evaluate() (ufl.referencevalue.ReferenceValue method), 155
evaluate() (ufl.restriction.Restricted method), 155
evaluate() (ufl.SpatialCoordinate method), 166
evaluate() (ufl.tensors.ComponentTensor method), 158
evaluate() (ufl.tensors.ListTensor method), 158
evaluate() (ufl.variable.Variable method), 160
execute_ufl_code() (in module ufl.algorithms.formfiles), 67
Expr (class in ufl.classes), 110
Expr (class in ufl.mathfunctions), 147
exp() (in module ufl), 177
exp() (in module ufl.core), 152
exp() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
expand_compounds() (in module ufl.algorithms), 76
expand_compounds() (in module ufl.algorithms.expand_compounds), 65
expand_derivatives() (in module ufl.algorithms), 75
expand_derivatives() (in module ufl.algorithms.ad), 46
expand_indices() (in module ufl.algorithms), 75
expand_indices() (in module ufl.algorithms.expand_indices), 66
expecting_expr() (in module ufl.assertions), 86
expecting_instance() (in module ufl.assertions), 86
expecting_python_scalar() (in module ufl.assertions), 86
expecting_true_ufl_scalar() (in module ufl.assertions), 86
Expr (class in ufl.classes), 89
Index
is_cellwise_constant() (ufl.FacetNormal method), 167
is_cellwise_constant() (ufl.finiteelement.enrichedelement.EnrichedElement method), 78
is_cellwise_constant() (ufl.finiteelement.enrichedelement.NodalEnrichedElement method), 79
is_cellwise_constant() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
is_cellwise_constant() (ufl.finiteelement.mixedelement.MixedElement method), 81
is_cellwise_constant() (ufl.finiteelement.restrictedelement.RestrictedElement method), 83
is_cellwise_constant() (ufl.FiniteElementBase method), 169
is_cellwise_constant() (ufl.geometry.CellCoordinate method), 136
is_cellwise_constant() (ufl.geometry.CellEdgeVectors method), 136
is_cellwise_constant() (ufl.geometry.CellFacetJacobian method), 137
is_cellwise_constant() (ufl.geometry.CellFacetJacobianDeterminant method), 137
is_cellwise_constant() (ufl.geometry.CellFacetJacobianInverse method), 137
is_cellwise_constant() (ufl.geometry.CellOrigin method), 138
is_cellwise_constant() (ufl.geometry.CellVertices method), 138
is_cellwise_constant() (ufl.geometry.FacetCoordinate method), 139
is_cellwise_constant() (ufl.geometry.FacetEdgeVectors method), 139
is_cellwise_constant() (ufl.geometry.FacetJacobian method), 140
is_cellwise_constant() (ufl.geometry.FacetJacobianDeterminant method), 140
is_cellwise_constant() (ufl.geometry.FacetJacobianInverse method), 140
is_cellwise_constant() (ufl.geometry.FacetNormal method), 141
is_cellwise_constant() (ufl.geometry.GeometricQuantity method), 141
is_cellwise_constant() (ufl.geometry.Jacobian method), 165
is_cellwise_constant() (ufl.geometry.JacobianDeterminant method), 165
is_cellwise_constant() (ufl.geometry.JacobianInverse method), 165
is_cellwise_constant() (ufl.geometry.QuadratureWeight method), 142
is_cellwise_constant() (ufl.geometry.ReferenceCellEdgeVectors method), 142
is_cellwise_constant() (ufl.geometry.ReferenceFacetEdgeVectors method), 143
is_cellwise_constant() (ufl.geometry.SpatialCoordinate method), 143
is_cellwise_constant() (ufl.Jacobian method), 168
is_cellwise_constant() (ufl.JacobianDeterminant method), 168
is_cellwise_constant() (ufl.JacobianInverse method), 168
is_cellwise_constant() (ufl.MixedElement method), 170
is_cellwise_constant() (ufl.NodalEnrichedElement method), 171
is_cellwise_constant() (ufl.RestrictedElement method), 171
is_cellwise_constant() (ufl.SpatialCoordinate method), 166
is_cellwise_constant() (ufl.variable.Label method), 159
is_globally_constant() (in module ufl.checks), 88
is_piecewise_linear_simplex_domain() (ufl.classes.Mesh method), 118
is_piecewise_linear_simplex_domain() (ufl.classes.MeshView method), 118
is_piecewise_linear_simplex_domain() (ufl.classes.TensorProductMesh method), 118
is_piecewise_linear_simplex_domain() (ufl.domain.Mesh method), 130
is_piecewise_linear_simplex_domain() (ufl.domain.MeshView method), 130
is_piecewise_linear_simplex_domain() (ufl.domain.TensorProductMesh method), 130
is_piecewise_linear_simplex_domain() (ufl.Mesh method), 165
is_piecewise_linear_simplex_domain() (ufl.MeshView method), 166
is_piecewise_linear_simplex_domain() (ufl.TensorProductMesh method), 166
is_post_handler() (in module ufl.algorithms.transformer), 73
is_python_scalar() (in module ufl.checks), 88
is_scalar_constant_expression() (in module ufl.checks), 88
is_simplex() (ufl.AbstractCell method), 164
is_simplex() (ufl.Cell method), 165
is_simplex() (ufl.cell.AbstractCell method), 87
is_simplex() (ufl.cell.Cell method), 87
is_simplex() (ufl.cell.TensorProductCell method), 88
is_simplex() (ufl.classes.AbstractCell method), 112
is_simplex() (ufl.classes.Cell method), 112
is_simplex() (ufl.classes.TensorProductCell method), 113
is_true_ufl_scalar() (in module ufl.checks), 88
iter_expressions() (in module ufl.algorithms.traversal), 73
J

Jacobian (class in ufl), 168
Jacobian (class in ufl.classes), 92
Jacobian (class in ufl.geometry), 141
jacobian() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
jacobian() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
jacobian_determinant() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
jacobian_inverse() (ufl.algorithms.apply_derivatives.GradRuleset method), 52
jacobian_inverse() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
JacobianDeterminant (class in ufl), 168
JacobianDeterminant (class in ufl.classes), 94
JacobianDeterminant (class in ufl.geometry), 141
JacobianInverse (class in ufl), 168
JacobianInverse (class in ufl.classes), 95
JacobianInverse (class in ufl.geometry), 141
join_domains() (in module ufl.domain), 130
jump() (in module ufl), 180
jump() (in module ufl.operators), 152

L

Label (class in ufl.classes), 101
Label (class in ufl.variable), 159
label() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
label() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
label() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
label() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
label() (ufl.classes.Variable method), 102
label() (ufl.variable.Variable method), 160
LE (class in ufl.classes), 108
LE (class in ufl.conditional), 124
le() (in module ufl), 178
le() (in module ufl.operators), 152
le() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
lhs() (in module ufl), 183
lhs() (in module ufl.formoperators), 134
linear_indexed_type() (ufl.algorithms.check_arities.ArityChecker method), 58
linear_indexed_type() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
list_tensor() (ufl.algorithms.apply_derivatives.GradRuleset method), 52
list_tensor() (ufl.algorithms.check_arities.ArityChecker method), 58
list_tensor() (ufl.algorithms.expand_indices.IndexExpander method), 66
list_tensor() (ufl.algorithms.expand_indices.IndexExpander method), 66
list_tensor() (ufl.algorithms.formtransformations.PartExtractor method), 68
ListTensor (class in ufl.operators), 100
ListTensor (class in ufl.operators), 100
Ln (class in ufl.classes), 110
Ln (class in ufl.mathfunctions), 147
ln() (in module ufl), 177
ln() (in module ufl.operators), 152
ln() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
load_forms() (in module ufl.algorithms.formfiles), 67
load_ufl_file() (in module ufl.algorithms), 74
load_ufl_file() (in module ufl.algorithms.formfiles), 67
log() (ufl.log.Logger method), 146
Logger (class in ufl.log), 145
LowerCompoundAlgebra (class in ufl.algorithms.apply_algebra_lowering), 47
LT (class in ufl.classes), 109
LT (class in ufl.conditional), 124
lt() (in module ufl), 178
lt() (in module ufl.operators), 152
lt() (ufl.algorithms.comparison_checker.CheckComparisons method), 60

M

map_integrand_dags() (in module ufl.algorithms.map_integrands), 70
map_integrands() (in module ufl.algorithms.map_integrands), 70
mapping() (ufl.BrokenElement method), 173
mapping() (ufl.classes.BrokenElement method), 173
mapping() (ufl.classes.FiniteElementBase method), 113
mapping() (ufl.classes.HCurlElement method), 117
mapping() (ufl.classes.HDivElement method), 117
mapping() (ufl.classes.MixedElement method), 114
mapping() (ufl.classes.RestrictedElement method), 116
mapping() (ufl.classes.TensorProductElement method), 116
mapping() (ufl.classes.TensorProductElement method), 116
mapping() (ufl.FiniteElement method), 169
mapping() (ufl.finiteelement.brokenelement.BrokenElement method), 77
mapping() (ufl.finiteelement.enrichedelement.EnrichedElementBase method), 78
mapping() (ufl.FiniteElement method), 79
MinFacetEdgeLength (class in ufl), 167
MinFacetEdgeLength (class in ufl.classes), 97
MinFacetEdgeLength (class in ufl.geometry), 142
MinValue (class in ufl.classes), 109
MinValue (class in ufl.conditional), 125
MixedElement (class in ufl), 169
MixedElement (class in ufl.classes), 114
MixedElement (class in ufl.finiteelement.mixedelement), 81
MixedFunctionSpace (class in ufl), 173
MixedFunctionSpace (class in ufl.classes), 119
MixedFunctionSpace (class in ufl.functionspace), 135
modulus() (ufl.classes.ComplexValue method), 99
modulus() (ufl.constantvalue.ComplexValue method), 126
multi_index() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
multi_index() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
multi_index() (ufl.algorithms.coordinate_derivative_helpers.CoordinateDerivativeIsOutermostChecker method), 61
multi_index() (ufl.algorithms.expand_indices.IndexExpander method), 64
multi_index() (ufl.algorithms.expand_indices.IndexExpander method), 66
multi_index() (ufl.algorithms.FormSplitter method), 76
multi_index() (ufl.algorithms.formsplitter.FormSplitter method), 67
multi_index() (ufl.algorithms.renumbering.IndexRenumberingTransformer method), 70
MultiFunction (class in ufl.algorithms), 75
MultiIndex (class in ufl.classes), 98

\( \nabla_d \)(in module ufl), 180
\( \nabla_d \)(in module ufl.operators), 152
\( \nabla_d \)(ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
\( \nabla_d \)(ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
\( \nabla_d \)(in module ufl), 180
\( \nabla_d \)(in module ufl.operators), 153
\( \nabla_d \)(ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
\( \nabla_d \)(ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
NablaDiv (class in ufl.classes), 108
NablaDiv (class in ufl.differentiation), 128
NablaGrad (class in ufl.classes), 107
NablaGrad (class in ufl.differentiation), 128
name (ufl.CellDiameter attribute), 167
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95
name (ufl.CellFaceNormal attribute), 167
name (ufl.CellOrientation attribute), 97
name (ufl.CellOrigin attribute), 91
name (ufl.CellVertices attribute), 167
name (ufl.CellVolume attribute), 96
name (ufl.Circumradius attribute), 167
name (ufl.CellDiameter attribute), 97
name (ufl.CellEdgeVectors attribute), 94
name (ufl.CellFacetJacobian attribute), 93
name (ufl.CellFacetJacobianDeterminant attribute), 95
name (ufl.CellFacetJacobianInverse attribute), 95

210 Index
<table>
<thead>
<tr>
<th>Method/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub_cells() (ufl.classes.TensorProductCell method)</td>
<td>113</td>
</tr>
<tr>
<td>sub_cells() (ufl.TensorProductCell method)</td>
<td>165</td>
</tr>
<tr>
<td>sub_element() (ufl.classes.RestrictedElement method)</td>
<td>116</td>
</tr>
<tr>
<td>sub_element() (ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>83</td>
</tr>
<tr>
<td>sub_elements() (ufl.classes.FiniteElementBase method)</td>
<td>114</td>
</tr>
<tr>
<td>sub_elements() (ufl.classes.MixedElement method)</td>
<td>115</td>
</tr>
<tr>
<td>sub_elements() (ufl.classes.RestrictedElement method)</td>
<td>116</td>
</tr>
<tr>
<td>sub_elements() (ufl.classes.TensorProductElement method)</td>
<td>117</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.finiteelementbase.FiniteElementBase method)</td>
<td>80</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.mixedelement.MixedElement method)</td>
<td>82</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>83</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.tensorproductelement.TensorProductElement method)</td>
<td>84</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.finiteelementbase.FiniteElementBase method)</td>
<td>169</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.mixedelement.MixedElement method)</td>
<td>170</td>
</tr>
<tr>
<td>sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>172</td>
</tr>
<tr>
<td>sub_elements_with_mappings() (in module ufl.algorithms.apply_function_pullbacks)</td>
<td>53</td>
</tr>
<tr>
<td>sub_forms_by_domain() (in module ufl.form)</td>
<td>133</td>
</tr>
<tr>
<td>sub_measures() (ufl.classes.MeasureProduct method)</td>
<td>120</td>
</tr>
<tr>
<td>sub_measures() (ufl.measure.MeasureProduct method)</td>
<td>149</td>
</tr>
<tr>
<td>subdomain_data() (ufl.classes.Form method)</td>
<td>122</td>
</tr>
<tr>
<td>subdomain_data() (ufl.classes.Integral method)</td>
<td>121</td>
</tr>
<tr>
<td>subdomain_data() (ufl.classes.Measure method)</td>
<td>120</td>
</tr>
<tr>
<td>subdomain_data() (ufl.Form method)</td>
<td>181</td>
</tr>
<tr>
<td>subdomain_data() (ufl.form.Form method)</td>
<td>133</td>
</tr>
<tr>
<td>subdomain_data() (ufl.integral.Integral method)</td>
<td>182</td>
</tr>
<tr>
<td>subdomain_data() (ufl.measure.Measure method)</td>
<td>145</td>
</tr>
<tr>
<td>subdomain_data() (ufl.measure.Measure method)</td>
<td>182</td>
</tr>
<tr>
<td>subdomain_id (ufl.algorithms.domain_analysis.IntegralData attribute)</td>
<td>61</td>
</tr>
<tr>
<td>subdomain_id() (ufl.classes.Integral method)</td>
<td>121</td>
</tr>
<tr>
<td>subdomain_id() (ufl.classes.Measure method)</td>
<td>120</td>
</tr>
<tr>
<td>subdomain_id() (ufl.pixel.Pixel method)</td>
<td>182</td>
</tr>
<tr>
<td>subdomain_id() (ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>145</td>
</tr>
<tr>
<td>subdomain_id() (ufl.measure.Measure method)</td>
<td>149</td>
</tr>
<tr>
<td>Sum (class in ufl.algebra)</td>
<td>85</td>
</tr>
<tr>
<td>Sum (class in ufl.classes)</td>
<td>102</td>
</tr>
<tr>
<td>sum() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method)</td>
<td>51</td>
</tr>
<tr>
<td>sum() (ufl.algorithms.check_arities.ArityChecker method)</td>
<td>58</td>
</tr>
<tr>
<td>sum() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method)</td>
<td>65</td>
</tr>
<tr>
<td>sum() (ufl.algorithms.formtransformations.PartExtrracter method)</td>
<td>68</td>
</tr>
<tr>
<td>Sym (class in ufl.classes)</td>
<td>105</td>
</tr>
<tr>
<td>Sym (class in ufl.tensoralgebra)</td>
<td>157</td>
</tr>
<tr>
<td>sym() (in module ufl)</td>
<td>177</td>
</tr>
<tr>
<td>sym() (in module ufl.operators)</td>
<td>153</td>
</tr>
<tr>
<td>Symmetry (class in ufl.classes)</td>
<td>110</td>
</tr>
<tr>
<td>Symmetry (class in ufl.functions)</td>
<td>115</td>
</tr>
<tr>
<td>Symmetry (class in ufl.finiteelement.finiteelementbase.FiniteElementBase method)</td>
<td>80</td>
</tr>
<tr>
<td>Symmetry (class in ufl.finiteelement.mixedelement.MixedElement method)</td>
<td>82</td>
</tr>
<tr>
<td>Symmetry (class in ufl.finiteelement.mixedelement.TensorElement method)</td>
<td>82</td>
</tr>
<tr>
<td>Symmetry (class in ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>83</td>
</tr>
<tr>
<td>Symmetry (ufl.finiteelement.finiteelementbase.FiniteElementBase method)</td>
<td>169</td>
</tr>
<tr>
<td>Symmetry (ufl.finiteelement.mixedelement.MixedElement method)</td>
<td>170</td>
</tr>
<tr>
<td>Symmetry (ufl.finiteelement.restrictedelement.RestrictedElement method)</td>
<td>172</td>
</tr>
<tr>
<td>Symmetry (ufl.finiteelement.tensorproductelement.TensorProductElement method)</td>
<td>171</td>
</tr>
<tr>
<td>system() (in module ufl)</td>
<td>184</td>
</tr>
<tr>
<td>system() (in module ufl.form)</td>
<td>135</td>
</tr>
<tr>
<td>T (ufl.classes.Expr attribute)</td>
<td>89</td>
</tr>
<tr>
<td>Tan (class in ufl.classes)</td>
<td>110</td>
</tr>
<tr>
<td>Tan (class in ufl.mathfunctions)</td>
<td>148</td>
</tr>
<tr>
<td>tan() (in module ufl)</td>
<td>178</td>
</tr>
<tr>
<td>tan() (in module ufl.operators)</td>
<td>154</td>
</tr>
<tr>
<td>tan() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method)</td>
<td>51</td>
</tr>
<tr>
<td>Tanh (class in ufl.classes)</td>
<td>110</td>
</tr>
<tr>
<td>Tanh (class in ufl.mathfunctions)</td>
<td>148</td>
</tr>
<tr>
<td>tanh() (in module ufl)</td>
<td>178</td>
</tr>
<tr>
<td>tanh() (in module ufl.operators)</td>
<td>154</td>
</tr>
<tr>
<td>tanh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method)</td>
<td>51</td>
</tr>
<tr>
<td>tear() (in module ufl)</td>
<td>62</td>
</tr>
<tr>
<td>TensorConstant() (in module ufl)</td>
<td>175</td>
</tr>
</tbody>
</table>
Index
ufl_free_indices (ufl.tensoralgebra.Deviatoric attribute), 157
ufl_free_indices (ufl.tensoralgebra.Dot attribute), 157
ufl_free_indices (ufl.tensoralgebra.Inner attribute), 157
ufl_free_indices (ufl.tensoralgebra.Inverse attribute), 157
ufl_free_indices (ufl.tensoralgebra.Outer attribute), 157
ufl_free_indices (ufl.tensoralgebra.Skew attribute), 157
ufl_free_indices (ufl.tensoralgebra.Sym attribute), 157
ufl_free_indices (ufl.tensoralgebra.Trace attribute), 157
ufl_free_indices (ufl.tensoralgebra.Transposed attribute), 158
ufl_free_indices (ufl.tensors.ComponentTensor attribute), 158
ufl_free_indices (ufl.tensors.ListTensor attribute), 158
ufl_free_indices (ufl.variable.Label attribute), 159
ufl_free_indices (ufl.variable.Variable attribute), 160
ufl_function_space() (ufl.Argument method), 174
ufl_function_space() (ufl.argument.Argument method), 86
ufl_function_space() (ufl.classes.Argument method), 101
ufl_function_space() (ufl.classes.Coefficient method), 101
ufl_function_space() (ufl.classes.Coeficient method), 101
ufl_function_space() (ufl.classes.Coefficient method), 175
ufl_function_space() (ufl.coefficient.Coefficient method), 122
ufl_id() (ufl.classes.Mesh method), 118
ufl_id() (ufl.classes.MeshView method), 118
ufl_id() (ufl.classes.TensorProductMesh method), 118
ufl_id() (ufl.domain.Mesh method), 130
ufl_id() (ufl.domain.MeshView method), 130
ufl_id() (ufl.domain.TensorProductMesh method), 130
ufl_id() (ufl.Mesh method), 166
ufl_id() (ufl.MeshView method), 166
ufl_id() (ufl.classes.TensorProductMesh method), 166
ufl_index_dimensions (ufl.classes.Abs attribute), 84
ufl_index_dimensions (ufl.classes.Conj attribute), 84
ufl_index_dimensions (ufl.classes.Div attribute), 84
ufl_index_dimensions (ufl.classes.Division attribute), 84
ufl_index_dimensions (ufl.classes.Denominator attribute), 85
ufl_index_dimensions (ufl.classes.Product attribute), 85
ufl_index_dimensions (ufl.classes.Real attribute), 85
ufl_index_dimensions (ufl.classes.Sum attribute), 85
ufl_index_dimensions (ufl.averaging.CellAvg attribute), 87
ufl_index_dimensions (ufl.averaging.FacetAvg attribute), 87
ufl_index_dimensions (ufl.classes.Abs attribute), 103
ufl_index_dimensions (ufl.classes.Atan2 attribute), 111
ufl_index_dimensions (ufl.classes.BesselFunction attribute), 111
ufl_index_dimensions (ufl.classes.CellAvg attribute), 111
ufl_index_dimensions (ufl.classes.CoefficientDerivative attribute), 106
ufl_index_dimensions (ufl.classes.Cofactor attribute), 105
ufl_index_dimensions (ufl.classes.ComponentTensor attribute), 100
ufl_index_dimensions (ufl.classes.Condition attribute), 108
ufl_index_dimensions (ufl.classes.Conditional attribute), 109
ufl_index_dimensions (ufl.classes.Conj attribute), 103
ufl_index_dimensions (ufl.classes.CoordinateDerivative attribute), 107
ufl_index_dimensions (ufl.classes.Cross attribute), 104
ufl_index_dimensions (ufl.classes.Curl attribute), 108
ufl_index_dimensions (ufl.classes.Determinant attribute), 105
ufl_index_dimensions (ufl.classes.Deviatoric attribute), 105
ufl_index_dimensions (ufl.classes.Div attribute), 107
ufl_index_dimensions (ufl.classes.Division attribute), 103
ufl_index_dimensions (ufl.classes.Dot attribute), 104
ufl_index_dimensions (ufl.classes.ExprList attribute), 106
ufl_index_dimensions (ufl.classes.ExprMapping attribute), 106
ufl_index_dimensions (ufl.classes.FacetAvg attribute), 111
ufl_index_dimensions (ufl.classes.Grad attribute), 107
ufl_index_dimensions (ufl.classes.Imag attribute), 104
ufl_index_dimensions (ufl.classes.Indexed attribute), 100
ufl_index_dimensions (ufl.classes.IndexSum attribute), 105
ufl_index_dimensions (ufl.classes.Inner attribute), 104
ufl_index_dimensions (ufl.classes.Inverse attribute), 105
ufl_index_dimensions (ufl.classes.Label attribute), 102
ufl_index_dimensions (ufl.classes.ListTensor attribute), 100
ufl_index_dimensions (ufl.classes.MathFunction attribute), 110
ufl_index_dimensions (ufl.classes.MaxValue attribute), 110
ufl_index_dimensions (ufl.classes.MinValue attribute), 109
ufl_index_dimensions (ufl.classes.MultiIndex attribute), 98
ufl_index_dimensions (ufl.classes.NablaDiv attribute), 108
ufl_index_dimensions (ufl.classes.NablaGrad attribute), 108
ufl_index_dimensions (ufl.classes.Outer attribute), 104
ufl_index_dimensions (ufl.classes.Power attribute), 103
ufl_index_dimensions (ufl.classes.Product attribute), 103
ufl_index_dimensions (ufl.classes.Real attribute), 103
ufl_index_dimensions (ufl.classes.RealValue attribute), 99
ufl_index_dimensions (ufl.classes.ReferenceCurl attribute), 108
ufl_index_dimensions (ufl.classes.ReferenceDiv attribute), 108
ufl_index_dimensions (ufl.classes.ReferenceDiv attribute), 108
Index 223
ufl_operands (ufl.classes.Operator attribute), 98
ufl_operands (ufl.classes.Terminal attribute), 90
ufl_shape (ufl.algebra.Abs attribute), 84
ufl_shape (ufl.algebra.Conj attribute), 84
ufl_shape (ufl.algebra.Division attribute), 84
ufl_shape (ufl.algebra.Imag attribute), 84
ufl_shape (ufl.algebra.Power attribute), 85
ufl_shape (ufl.algebra.Product attribute), 85
ufl_shape (ufl.algebra.Real attribute), 85
ufl_shape (ufl.algebra.Sum attribute), 85
ufl_shape (ufl.Argument attribute), 174
ufl_shape (ufl.argument.Argument attribute), 86
ufl_shape (ufl.averaging.CellAvg attribute), 87
ufl_shape (ufl.averaging.FacetAvg attribute), 87
ufl_shape (ufl.CellNormal attribute), 167
ufl_shape (ufl.classes.Abs attribute), 103
ufl_shape (ufl.classes.Argument attribute), 101
ufl_shape (ufl.classes.Atan2 attribute), 111
ufl_shape (ufl.classes.BesselFunction attribute), 111
ufl_shape (ufl.classes.CellAvg attribute), 111
ufl_shape (ufl.classes.CellCoordinate attribute), 91
ufl_shape (ufl.classes.CellEdgeVectors attribute), 94
ufl_shape (ufl.classes.CellFacetJacobian attribute), 93
ufl_shape (ufl.classes.CellFacetJacobianInverse attribute), 95
ufl_shape (ufl.classes.CellFacetOrigin attribute), 92
ufl_shape (ufl.classes.CellNormal attribute), 96
ufl_shape (ufl.classes.CellOrigin attribute), 92
ufl_shape (ufl.classes.CellVertices attribute), 94
ufl_shape (ufl.classes.Coefficient attribute), 101
ufl_shape (ufl.classes.CoefficientDerivative attribute), 106
ufl_shape (ufl.classes.Cofactor attribute), 105
ufl_shape (ufl.classes.ComponentTensor attribute), 100
ufl_shape (ufl.classes.Condition attribute), 108
ufl_shape (ufl.classes.Conditional attribute), 109
ufl_shape (ufl.classes.Conj attribute), 103
ufl_shape (ufl.classes.Constant attribute), 101
ufl_shape (ufl.classes.CoordinateDerivative attribute), 107
ufl_shape (ufl.classes.Cross attribute), 104
ufl_shape (ufl.classes.Curl attribute), 108
ufl_shape (ufl.classes.Determinant attribute), 105
ufl_shape (ufl.classes.Deviatoric attribute), 105
ufl_shape (ufl.classes.Div attribute), 107
ufl_shape (ufl.classes.Division attribute), 103
ufl_shape (ufl.classes.Dot attribute), 104
ufl_shape (ufl.classes.ExprList attribute), 106
ufl_shape (ufl.classes.ExprMapping attribute), 106
ufl_shape (ufl.classes.FacetAvg attribute), 111
ufl_shape (ufl.classes.FacetCoordinate attribute), 91
ufl_shape (ufl.classes.FacetEdgeVectors attribute), 94
ufl_shape (ufl.classes.FacetJacobian attribute), 93
ufl_shape (ufl.classes.FacetJacobianInverse attribute), 95
ufl_shape (ufl.classes.FacetNormal attribute), 96
ufl_shape (ufl.classes.FacetOrigin attribute), 92
ufl_shape (ufl.classes.GeometricQuantity attribute), 90
ufl_shape (ufl.classes.Grad attribute), 107
ufl_shape (ufl.classes.Identity attribute), 100
ufl_shape (ufl.classes.Imag attribute), 104
ufl_shape (ufl.classes.Indexed attribute), 100
ufl_shape (ufl.classes.IndexSum attribute), 105
ufl_shape (ufl.classes.Inner attribute), 104
ufl_shape (ufl.classes.Inverse attribute), 105
ufl_shape (ufl.classes.Jacobian attribute), 92
ufl_shape (ufl.classes.JacobianInverse attribute), 95
ufl_shape (ufl.classes.Label attribute), 102
ufl_shape (ufl.classes.ListTensor attribute), 100
ufl_shape (ufl.classes.MathFunction attribute), 110
ufl_shape (ufl.classes.MaxValue attribute), 110
ufl_shape (ufl.classes.MinValue attribute), 109
ufl_shape (ufl.classes.MultiIndex attribute), 98
ufl_shape (ufl.classes.NablaDiv attribute), 108
ufl_shape (ufl.classes.NablaGrad attribute), 108
ufl_shape (ufl.classes.Outer attribute), 104
ufl_shape (ufl.classes.PermutationSymbol attribute), 100
ufl_shape (ufl.classes.Power attribute), 103
ufl_shape (ufl.classes.Product attribute), 103
ufl_shape (ufl.classes.Real attribute), 103
ufl_shape (ufl.classes.RealValue attribute), 99
ufl_shape (ufl.classes.ReferenceCellEdgeVectors attribute), 93
ufl_shape (ufl.classes.ReferenceCurl attribute), 108
ufl_shape (ufl.classes.ReferenceDiv attribute), 107
ufl_shape (ufl.classes.ReferenceFacetEdgeVectors attribute), 93
ufl_shape (ufl.classes.ReferenceGrad attribute), 107
ufl_shape (ufl.classes.ReferenceNormal attribute), 96
ufl_shape (ufl.classes.ReferenceValue attribute), 112
ufl_shape (ufl.classes.Restricted attribute), 106
ufl_shape (ufl.classes.ScalarValue attribute), 99
ufl_shape (ufl.classes.Skew attribute), 105
ufl_shape (ufl.classes.SpatialCoordinate attribute), 91
ufl_shape (ufl.classes.Sum attribute), 102
ufl_shape (ufl.classes.Sym attribute), 105
ufl_shape (ufl.classes.Trace attribute), 104
ufl_shape (ufl.classes.Transposed attribute), 104
ufl_shape (ufl.classes.Variable attribute), 102
ufl_shape (ufl.classes.VariableDerivative attribute), 107
ufl_shape (ufl.classes.Zero attribute), 99
ufl_shape (ufl.classes.Coefficient attribute), 175
ufl_shape (ufl.coefficient.Coefficient attribute), 123
ufl_shape (ufl.conditional.Condition attribute), 124
ufl_shape (ufl.conditional.Conditioned attribute), 124
ufl_shape (ufl.conditional.MaxValue attribute), 125
ufl_shape (ufl.conditional.MinValue attribute), 125
ufl_shape (ufl.classes.Constant attribute), 125
ufl_shape (ufl.constantvalue.Identity attribute), 126
ufl_shape (ufl.constantvalue.PermutationSymbol attribute), 126
ufl_shape (ufl.constantvalue.RealValue attribute), 127
ufl_shape (ufl.constantvalue.ScalarValue attribute), 127
ufl_shape (ufl.constantvalue.Zero attribute), 127
ufl_shape (ufl.differentiation.CoefficientDerivative attribute), 128
ufl_shape (ufl.differentiation.CoordinateDerivative attribute), 128
ufl_shape (ufl.differentiation.Curl attribute), 128
ufl_shape (ufl.differentiation.Div attribute), 128
ufl_shape (ufl.differentiation.Grad attribute), 128
ufl_shape (ufl.differentiation.NablaDiv attribute), 128
ufl_shape (ufl.differentiation.NablaGrad attribute), 129
ufl_shape (ufl.differentiation.ReferenceCurl attribute), 129
ufl_shape (ufl.differentiation.ReferenceDiv attribute), 129
ufl_shape (ufl.differentiation.ReferenceGrad attribute), 129
ufl_shape (ufl.differentiation.VariableDerivative attribute), 129
ufl_shape (ufl.exprcontainers.ExprList attribute), 131
ufl_shape (ufl.exprcontainers.ExprMapping attribute), 131
ufl_shape (ufl.FacetNormal attribute), 167
ufl_shape (ufl.geometry.CellCoordinate attribute), 136
ufl_shape (ufl.geometry.CellEdgeVectors attribute), 136
ufl_shape (ufl.geometry.CellFacetJacobian attribute), 137
ufl_shape (ufl.geometry.CellFacetJacobianInverse attribute), 137
ufl_shape (ufl.geometry.CellFacetOrigin attribute), 137
ufl_shape (ufl.geometry.CellNormal attribute), 138
ufl_shape (ufl.geometry.CellNormalOrigin attribute), 138
ufl_shape (ufl.geometry.CellVertices attribute), 138
ufl_shape (ufl.geometry.CellFaceCoordinate attribute), 139
ufl_shape (ufl.geometry.CellFaceJacobian.attribute), 139
ufl_shape (ufl.geometry.CellFaceJacobianInverse attribute), 140
ufl_shape (ufl.geometry.CellJacobian attribute), 140
ufl_shape (ufl.geometry.CellJacobianInverse attribute), 140
ufl_shape (ufl.geometry.FacetNormal attribute), 140
ufl_shape (ufl.geometry.FacetOrigin attribute), 141
ufl_shape (ufl.geometry.GeometricQuantity attribute), 141
ufl_shape (ufl.geometry.Jacobian attribute), 141
ufl_shape (ufl.geometry.JacobianInverse attribute), 142
ufl_shape (ufl.geometry.ReferenceCellEdgeVectors attribute), 142
ufl_shape (ufl.geometry.ReferenceFacetEdgeVectors attribute), 143
ufl_shape (ufl.geometry.ReferenceNormal attribute), 143
ufl_shape (ufl.geometry.SpatialCoordinate attribute), 143
ufl_shape (ufl.Identity attribute), 175
ufl_shape (ufl.indexed.Indexed attribute), 144
ufl_shape (ufl.indexsum.IndexSum attribute), 145
ufl_shape (ufl.Jacobian attribute), 168
ufl_shape (ufl.JacobianInverse attribute), 168
ufl_shape (ufl.mathfunctions.Atan2 attribute), 147
ufl_shape (ufl.mathfunctions.BesselFunction attribute), 147
ufl_shape (ufl.mathfunctions.MathFunction attribute), 148
ufl_shape (ufl.PermutationSymbol attribute), 175
ufl_shape (ufl.referencevalue.ReferenceValue attribute), 155
ufl_shape (ufl.restriction.Restricted attribute), 155
ufl_shape (ufl.SpatialCoordinate attribute), 166
ufl_shape (ufl.tensors.Cofactor attribute), 156
ufl_shape (ufl.tensors.Cross attribute), 156
ufl_shape (ufl.tensors.Determinant attribute), 157
ufl_shape (ufl.tensors.Deviatoric attribute), 157
ufl_shape (ufl.tensors.Dot attribute), 157
ufl_shape (ufl.tensors.Inner attribute), 157
ufl_shape (ufl.tensors.Inverse attribute), 157
ufl_shape (ufl.tensors.Outer attribute), 157
ufl_shape (ufl.tensors.Skew attribute), 157
ufl_shape (ufl.tensors.Sym attribute), 158
ufl_shape (ufl.tensors.Trace attribute), 158
ufl_shape (ufl.tensors.Transposed attribute), 158
ufl_shape (ufl.tensors.SymmetricTensor attribute), 158
ufl_shape (ufl.tensors.ListTensor attribute), 158
ufl_shape (ufl.variable.Label attribute), 160
ufl_shape (ufl.variable.Variable attribute), 160
ufl_sub_space() (ufl.classes.MixedFunctionSpace method), 119
ufl_sub_space() (ufl.functions.MixedFunctionSpace method), 136
ufl_sub_spaces() (ufl.MixedFunctionSpace method), 174
ufl_sub_spaces() (ufl.classes.AbstractFunctionSpace method), 118
ufl_sub_spaces() (ufl.classes.FunctionSpace method), 119
ufl_sub_spaces() (ufl.classes.MixedFunctionSpace method), 119
ufl_sub_spaces() (ufl.classes.TensorProductFunctionSpace method), 119
ufl_sub_spaces() (ufl.FunctionSpace method), 173
ufl_sub_spaces() (ufl.functions.MixedFunctionSpace method), 135
ufl_sub_spaces() (ufl.functions.FunctionSpace method), 135
ufl_sub_spaces() (ufl.functions.TensorProductFunctionSpace method), 136
ufl_sub_spaces() (ufl.functions.MixedFunctionSpace method), 174
UFLException, 164
undefined() (ufl.algorithms.MultiFunction method), 75