Unified Form Language (UFL) Documentation

Release 2019.1.0

FEniCS Project

Apr 17, 2019

Contents

1 Documentation **3**

Python Module Index **189**

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

CHAPTER 1

Documentation

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

```
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:

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

Here the form a depends on the *form arguments* $\mathbf{v} = (v_1, \ldots, v_r)$ and the *form coefficients* $\mathbf{w} = (w_1, \ldots, w_n)$, 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 Ω (`dx`, cell integral);
- the boundary $\partial\Omega$ of Ω (`ds`, exterior facet integral);
- the set of interior facets Γ (`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 Ω. This may be expressed as:

```
a = v*dx
```

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

```
a = v*ds.
```

Alternatively, measures can be redefined to represent numbered subsets of a domain, such that a form 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

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

Note: The domain Ω, 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:

```
cell = Cell(shape, 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:

```
# 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

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

Some examples:

```
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

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

Some examples:

```
# 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

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

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

Examples:

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

Mixed elements

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

General notation for an arbitrary number of subelements:

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

Shorthand notation for two subelements:

```
element = element1 * element2
```

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

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

represents `(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:

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

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

EnrichedElement

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

Example: The Mini element can be constructed as

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

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

Form arguments

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

Basis functions

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

```
v = Argument(element)
```

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

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

```
v, q = Arguments(TH)
u, p = Arguments(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 = Argument(TH)
v, q = split(up)
```

This is equivalent to the previous use of `Arguments`:

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

For convenience, `TestFunction` and `TrialFunction` are special instances of `Argument` with the property that a `TestFunction` will always be the first argument in a form and `TrialFunction` will always be the second argument in a form (order of declaration does not matter). Their usage is otherwise the same as for `Argument`:

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

Meshes and function spaces

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

Coefficient functions

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

```
f = Coefficient(element)
```

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

`Coefficient` is used to represent user-defined functions, including, e.g., source terms, body forces, variable coefficients and stabilization terms. UFL treats each `Coefficient` as a linear combination of unknown basis functions

with unknown coefficients, that is, UFL knows nothing about the concrete basis functions of the element and nothing about the value of the function.

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

```
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:

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

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

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

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

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

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

```
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)`:

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

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

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

```
n = 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:

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

# d x d identiy 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]*w[i]`: $\sum_{i=0}^{n-1} v_i w_i = \mathbf{v} \cdot \mathbf{w}$
- `Dx(v, i)*Dx(w, i)`: $\sum_{i=0}^{d-1} \frac{\partial v}{\partial x_i} \frac{\partial w}{\partial x_i} = \nabla v \cdot \nabla w$
- `Dx(v[i], i)`: $\sum_{i=0}^{d-1} \frac{\partial v_i}{\partial x_i} = \nabla \cdot v$
- `Dx(v[i], j)*Dx(w[i], j)`: $\sum_{i=0}^{n-1} \sum_{j=0}^{d-1} \frac{\partial v_i}{\partial x_j} \frac{\partial w_i}{\partial x_j} = \nabla \mathbf{v} : \nabla \mathbf{w}$

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 $\mathbf{i}_i$ for $i \in 0, \ldots, d-1$, we can define the dot product of any two basis functions as

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

where δ_{ij} is the Kronecker delta

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

A rank 1 tensor (vector) quantity $\mathbf{v}$ can be represented in terms of unit vectors and its scalar components in that basis. In tensor algebra it is common to assume implicit summation over indices repeated twice in a product:

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

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

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

This generalizes to tensors of arbitrary rank:

$$\begin{aligned}\mathcal{C} &= C_\iota \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}} \\ &\equiv \sum_{\iota_0} \cdots \sum_{\iota_{r-1}} C_\iota \mathbf{i}_{\iota_0} \otimes \cdots \otimes \mathbf{i}_{\iota_{r-1}},\end{aligned}$$

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

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

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

Defining indices

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

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

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

Each of these represents an `index range` determined by the context; if used to subscript a tensor-valued expression, the range is given by the shape of the expression, and if used to subscript a derivative, the range is given by the dimension d of the underlying shape of the finite element space. As we shall see below, indices can be a powerful tool when used to define forms in tensor notation.

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:

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

Note: Advanced usage

Note that in the old FFC notation, the definition

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

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

```
i = 0
```

Taking components of tensors

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

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

Basic indexing:

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

More advanced indexing:

- `A[i,0]`: component access, representing the scalar value of some component i of the first column of A
- `A[i,:]`: row access, representing some row i of A, i.e. rank(A[i,:]) == 1
- `A[:,j]`: column access, representing some column j of A, i.e. rank(A[:,j]) == 1
- `C[...,0]`: subtensor access, representing the subtensor of A with the last axis fixed, e.g., A[...,0] == A[:,0]
- `C[j,...]`: subtensor access, representing the subtensor of A with the 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 v_k to $\mathbf{v}$ or A_{ij} to $\mathbf{A}$, the following examples show how this is done:

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

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

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

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

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

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

$$\mathbf{v} = \mathbf{i}_0 + 2\mathbf{i}_1 + 3\mathbf{i}_2,$$
$$\mathbf{A} = \begin{bmatrix} u_0 & 0 \\ 0 & u_1 \end{bmatrix},$$
$$\mathbf{B} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}.$$

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

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

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

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

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

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

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.

- `abs(f)`: the absolute value of f.

- `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:

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

The definition of the transpose is

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

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

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

tr

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

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

The definition of the trace is

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

dot

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

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

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

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

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

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

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

An example with two vectors

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

An example with a tensor of rank two

$$\begin{aligned} \mathbf{A} \cdot \mathbf{B} &= (A_{ij} \mathbf{i}_i \mathbf{i}_j) \cdot (B_{kl} \mathbf{i}_k \mathbf{i}_l) \\ &= (A_{ij} B_{kl}) \mathbf{i}_i (\mathbf{i}_j \cdot \mathbf{i}_k) \mathbf{i}_l \\ &= (A_{ij} B_{kl} \delta_{jk}) \mathbf{i}_i \mathbf{i}_l \\ &= A_{ik} B_{kl} \mathbf{i}_i \mathbf{i}_l. \end{aligned}$$

This is the same as a matrix-matrix multiplication.

An example with a vector and a tensor of rank two

$$\begin{aligned} \mathbf{v} \cdot \mathbf{A} &= (v_j \mathbf{i}_j) \cdot (A_{kl} \mathbf{i}_k \mathbf{i}_l) \\ &= (v_j A_{kl}) (\mathbf{i}_j \cdot \mathbf{i}_k) \mathbf{i}_l \\ &= (v_j A_{kl} \delta_{jk}) \mathbf{i}_l \\ &= v_k A_{kl} \mathbf{i}_l \end{aligned}$$

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 rank(a)+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 $\mathbf{A}$ and $\mathbf{B}$ are rank two tensors and $\mathcal{C}$ and $\mathcal{D}$ are rank 3 tensors their inner products are

$$\mathbf{A} : \mathbf{B} = A_{ij} B_{ij}^*$$
$$\mathcal{C} : \mathcal{D} = C_{ijk} D_{ijk}^*$$

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

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

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

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

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:

$$\mathbf{v} \otimes \mathbf{u} = v_i^* u_j \mathbf{i}_i \mathbf{i}_j,$$
$$\mathbf{v} \otimes \mathbf{B} = v_i^* B_{kl} \mathbf{i}_i \mathbf{i}_k \mathbf{i}_l,$$
$$\mathbf{A} \otimes \mathbf{B} = A_{ij}^* B_{kl} \mathbf{i}_i \mathbf{i}_j \mathbf{i}_k \mathbf{i}_l.$$

The outer product of vectors is often written simply as

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

which is what we have done with $\mathbf{i}_i \mathbf{i}_j$ 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:

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

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 = dev(A)
```

The definition is

$$\mathrm{dev}\mathbf{A} = \mathbf{A} - \frac{\mathbf{A}_{ii}}{d}\mathbf{I}$$

where d is the rank of matrix A and $\mathbf{I}$ is the identity matrix.

sym

The symmetric part of A can be written as

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

The definition is

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

skew

The skew symmetric part of A can be written as

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

The definition is

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

cofac

The cofactor of a matrix A can be written as

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

The definition is

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

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

inv

The inverse of matrix A can be written as

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

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

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:

```
# 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:

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

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

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

Note: *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 = grad(f)
df = div(f)
cf = curl(v)
rf = rot(f)
```

The operand `f` can have no free indices.

Gradient

The gradient of a scalar u is defined as

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

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

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

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

which, written componentwise, reads

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

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

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

where ι is a multi-index of length r.

In UFL, the following pairs of declarations are equivalent:

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

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

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

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

Divergence

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

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

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

In UFL, the following declarations are equivalent:

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

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

for a vector expression v and a tensor expression A.

Curl and rot

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

$$\mathrm{curl}(\mathbf{v}) = \nabla \times \mathbf{v} = (\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}).$$

Note: The *curl* or *rot* operator is only defined for vectors of length three.

In UFL, the following declarations are equivalent:

```
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:

```
# 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:

```
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. to 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('+')` and `v('-')`

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

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

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

f = Coefficient(element)

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

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

Jump: `jump(v)`

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

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

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

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

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

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

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

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

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

Average: `avg(v)`

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

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

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

Conditional Operators

Conditional

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

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

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

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

Conditions

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

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

User-defined operators

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

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

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

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

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

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

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

- `conj(f)` :: complex conjugate of `f`.
- `imag(f)` :: imaginary part of `f`.
- `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 permissable 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:

```
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:

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

To give a concrete example, these declarations are equivalent:

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

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

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

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

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:

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

which is equivalent to:

```
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:

```
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`:

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

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

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

As a slightly more complicated example, this formulation:

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

is equivalent to

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

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

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

which is equivalent to

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

Assume in the following examples that

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

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

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

which is equivalent to

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

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

```
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:

```
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`):

```
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:

```
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

```
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 `dx` has been declared as an integral over the interior of Ω and that both `i` and `j` have been declared as a free `Index`.

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

The mass matrix

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

$$a(v, u) = \int_\Omega v\, u\, dx,$$

which can be implemented in UFL as follows:

```
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:

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

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

(continues on next page)

(continued from previous page)

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

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

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

or like this:

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

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

Vector-valued Poisson

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

$$a(v, u) = \int_\Omega \nabla v : \nabla u \, dx,$$
$$L(v; f) = \int_\Omega v \cdot f \, dx,$$

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

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

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

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

Alternatively, index notation may be used like this:

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

or like this:

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

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

The strain-strain term of linear elasticity

The strain-strain term of linear elasticity,

$$a(v, u) = \int_\Omega \epsilon(v) : \epsilon(u) \, dx,$$

where

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

can be implemented as follows:

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

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

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

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

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

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

or like this:

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

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

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:

```
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:

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

or like this:

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

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

$$\frac{1}{k_n} \int_\Omega v\,(u_h^n - u_h^{n-1})\,dx + \int_\Omega c\,\nabla v \cdot \nabla u_h^n\,dx = \int_\Omega v\,f^n\,dx$$

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

$$a(v, u_h^n; c, k) = \int_\Omega v\,u_h^n\,dx + k_n \int_\Omega c\,\nabla v \cdot \nabla u_h^n\,dx,$$
$$L(v; u_h^{n-1}, f, k) = \int_\Omega v\,u_h^{n-1}\,dx + k_n \int_\Omega v\,f^n\,dx,$$

which can be implemented as follows:

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

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

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

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

Mixed formulation of Stokes

To solve Stokes' equations,

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

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

$$a((v, q), (u, p)) = \int_\Omega \nabla v : \nabla u - (\nabla \cdot v)\,p + q\,(\nabla \cdot u)\,dx,$$
$$L((v, q); f) = \int_\Omega v \cdot f\,dx\,.$$

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

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

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

f = Coefficient(P2)

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

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

Mixed formulation of Poisson

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

$$\begin{aligned} \sigma + \nabla u &= 0, \\ \nabla \cdot \sigma &= f. \end{aligned}$$

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

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

where

$$\begin{aligned} a((\tau, w), (\sigma, u)) &= \int_\Omega \tau \cdot \sigma - \nabla \cdot \tau\, u + w \nabla \cdot \sigma \, dx, \\ L((\tau, w); f) &= \int_\Omega w \cdot f \, dx \,. \end{aligned}$$

We may implement the corresponding forms in our form language using first order BDM H(div)-conforming elements for σ and piecewise constant L^2-conforming elements for u as follows:

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

element = BDM1 * DG0

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

f = 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

$$\begin{aligned} a(v, u; h) &= \int_\Omega \nabla v \cdot \nabla u \, dx \\ &\quad + \sum_S \int_S -\langle \nabla v \rangle \cdot [[u]]_n - [[v]]_n \cdot \langle \nabla u \rangle + (\alpha/h)[[v]]_n \cdot [[u]]_n \, dS \\ &\quad + \int_{\partial\Omega} -\nabla v \cdot [[u]]_n - [[v]]_n \cdot \nabla u + (\gamma/h) v u \, ds \\ L(v; f, g) &= \int_\Omega v f \, dx + \int_{\partial\Omega} v g \, ds \,. \end{aligned}$$

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

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

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

f = Coefficient(DG1)
g = Coefficient(DG1)
h = 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*dx + 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,

$$\begin{aligned} a(v, u; u_0) &= \int_\Omega (1 + u_0^2) \nabla v \cdot \nabla u \, dx + \int_\Omega 2 u_0 u \nabla v \cdot \nabla u_0 \, dx, \\ L(v; u_0, f) &= \int_\Omega v \, f \, dx - \int_\Omega (1 + u_0^2) \nabla v \cdot \nabla u_0 \, dx, \end{aligned}$$

can be implemented in a single form file as follows:

```
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

$$a(v, u) = \int_\Omega \mathrm{C}\nabla v \cdot \nabla u \, dx + \int_\Omega 2u_0 u \nabla v \cdot \nabla u_0 \, dx,$$
$$L(v; \sigma_0, f) = \int_\Omega v \, f \, dx - \int_\Omega \nabla v \cdot \sigma_0 \, dx \, .$$

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

```
# 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

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

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

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

and

```
# 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:

```
Iteration REF1      FE1      QE1
========= ====      ===      ===
1         6.3e-02   6.3e-02  6.3e-02
2         5.3e-04   5.3e-04  5.3e-04
3         3.7e-08   3.7e-08  3.7e-08
4         2.9e-16   2.9e-16  2.5e-16
```

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:

```
Iteration REF2      FE2      QE2
========= ====      ===      ===
```

(continues on next page)

(continued from previous page)

```
1          2.6e-01   3.9e-01  2.6e-01
2          1.1e-02   4.6e-02  1.1e-02
3          1.2e-05   1.1e-02  1.6e-05
4          1.1e-11   7.2e-04  9.1e-09
```

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

```
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

```
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)

(continued from previous page)

```
          Coefficient(FunctionSpace(Mesh(VectorElement('Lagrange', triangle, 1,␣
↪dim=2), -1), FiniteElement('Lagrange', triangle, 1)), 0)
        )
      )
Integral:
  integral type: exterior_facet
  subdomain id: everywhere
  integrand:
    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)
    )
```

Inspecting and manipulating the expression tree

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

Traversing expressions

iter_expressions

Example usage:

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

outputs:

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

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:

```
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))
```

(continues on next page)

(continued from previous page)

```
        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
```

$(v_h^0)(v_h^1)$

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:

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

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

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

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

element = FiniteElement("CG", triangle, 1)
f = 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.

$$v(x) = \phi_k(x), \qquad \phi_k \in V_h.$$

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

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

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

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

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

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

In addition, we need that

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

which in UFL can be represented as

$$w = \texttt{Coefficient(element)},$$
$$v = \texttt{Argument(element)},$$
$$\frac{dw}{dw_j} = v,$$

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[1] computational graph from the abstract expression tree.

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

The computational graph

Consider the expression

$$f = (a + b) * (c + d)$$

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

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

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

```
Coefficient Coefficient Coefficient Coefficient
          \     /             \     /
            Sum                 Sum
               \               /
                --- Product ---
```

If we instead have the expression

$$f = (a + b) * (a - b)$$

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

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

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

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

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

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

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

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

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

$$v_i \prec v_j, \quad \forall j > i,$$

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 = partition(G)
```

1.3 ufl package

1.3.1 Subpackages

ufl.algorithms package

Submodules

ufl.algorithms.ad module

Front-end for AD routines.

ufl.algorithms.ad.**expand_derivatives**(*form*, ***kwargs*)
Expand all derivatives of expr.

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

ufl.algorithms.analysis module

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

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

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.

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

ufl.algorithms.analysis.**extract_elements**(*form*)
Build sorted tuple of all elements used in form.

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

ufl.algorithms.analysis.**extract_type**(*a*, *ufl_type*)
Build a set of all objects of class ufl_type found in a. The argument a can be a Form, Integral or Expr.

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

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

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

ufl.algorithms.analysis.**sort_elements**(*elements*)
Sort elements so that any sub elements appear before the corresponding mixed elements. This is useful when sub elements need to be defined before the corresponding mixed elements.

The ordering is based on sorting a directed acyclic graph.

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

ufl.algorithms.apply_algebra_lowering module

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

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

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

altenative_dot(*o*, *a*, *b*)

alternative_inner(*o*, *a*, *b*)

cofactor(*o*, *A*)

cross(*o*, *a*, *b*)

curl(*o*, *a*)

determinant(*o*, *A*)

deviatoric(*o*, *A*)

div(*o*, *a*)

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

expr(*o*, **ops*)
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

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

inverse(*o*, *A*)

nabla_div(*o*, *a*)

nabla_grad(*o*, *a*)

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

skew(*o*, *A*)

sym(*o*, *A*)

trace(*o*, *A*)

transposed(*o*, *A*)

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

ufl.algorithms.apply_derivatives module

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

class ufl.algorithms.apply_derivatives.**CoordinateDerivativeRuleDispatcher**
Bases: ufl.corealg.multifunction.MultiFunction

coefficient_derivative(*o*)

coordinate_derivative(*o*)

derivative(*o*)

expr(*o*, **ops*)
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

grad(*o*)

reference_grad(*o*)

terminal(*o*)

class ufl.algorithms.apply_derivatives.**CoordinateDerivativeRuleset**(*coefficients*, *arguments*, *coefficient_derivatives*)
Bases: *ufl.algorithms.apply_derivatives.GenericDerivativeRuleset*

Apply AFD (Automatic Functional Differentiation) to expression.

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

D_w[v](e) = d/dtau e(w+tau v)|tau=0

where 'e' is a ufl form after pullback and w is a SpatialCoordinate.

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

coefficient(*o*)

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

grad(*o*)

jacobian(*o*)

reference_grad(*g*)

reference_value(*o*)

spatial_coordinate(*o*)

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](...) defined as

D_w[v](e) = 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_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*)
dX/dx = inv(dx/dX) = 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 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*)
dx/dx = I

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

argument(*o*)

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

cell_coordinate(*o*)
dX/dX = I

coefficient(*o*)

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

geometric_quantity(*o*)
dg/dX = 0 if piecewise constant, otherwise ReferenceGrad(g)

grad(*o*)

reference_grad(*o*)
Represent ref_grad(ref_grad(f)) as RefGrad(RefGrad(f)).

reference_value(*o*)

spatial_coordinate(*o*)
dx/dX = J

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

argument(*o*)
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*)
df/dv = Id if v is f else 0.

Note that if v = variable(f), df/dv is still 0, but if v == f, i.e. isinstance(v, Coefficient) == True, then df/dv == df/df = Id.

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

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

grad(*o*)
Variable derivative of a gradient of a terminal must be 0.

reference_grad(*o*)
Variable derivative of a gradient of a terminal must be 0.

reference_value(*o*)

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

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.

class ufl.algorithms.apply_function_pullbacks.**FunctionPullbackApplier**
Bases: ufl.corealg.multifunction.MultiFunction

expr(*o*, **ops*)
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

form_argument(*o*)

terminal(*t*)

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

@param expr: An Expr.

ufl.algorithms.apply_function_pullbacks.**apply_single_function_pullbacks**(*g*)

ufl.algorithms.apply_function_pullbacks.**create_nested_lists**(*shape*)

ufl.algorithms.apply_function_pullbacks.**reshape_to_nested_list**(*components*, *shape*)

ufl.algorithms.apply_function_pullbacks.**sub_elements_with_mappings**(*element*)
Return an ordered list of the largest subelements that have a defined mapping.

ufl.algorithms.apply_geometry_lowering module

Algorithm for lowering abstractions of geometric types.

This means replacing high-level types with expressions of mostly the Jacobian and reference cell data.

class ufl.algorithms.apply_geometry_lowering.**GeometryLoweringApplier**(*preserve_types=()*)
Bases: ufl.corealg.multifunction.MultiFunction

cell_coordinate(*o*)

cell_diameter(*o*)

cell_normal(*o*)

cell_volume(*o*)

circumradius(*o*)

expr(*o*, **ops*)
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

facet_area(*o*)

facet_cell_coordinate(*o*)

facet_jacobian(*o*)

facet_jacobian_determinant(*o*)

facet_jacobian_inverse(*o*)

facet_normal(*o*)

jacobian(*o*)

jacobian_determinant(*o*)

jacobian_inverse(*o*)

max_cell_edge_length(*o*)

max_facet_edge_length(*o*)

min_cell_edge_length(*o*)

min_facet_edge_length(*o*)

spatial_coordinate(*o*)

terminal(*t*)

ufl.algorithms.apply_geometry_lowering.**apply_geometry_lowering**(*form*, *preserve_types=()*)

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

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

@param form: An Expr or Form.

ufl.algorithms.apply_integral_scaling module

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

ufl.algorithms.apply_integral_scaling.**apply_integral_scaling**(*form*)

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

ufl.algorithms.apply_integral_scaling.**compute_integrand_scaling_factor**(*integral*)

Change integrand geometry to the right representations.

ufl.algorithms.apply_restrictions module

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

class ufl.algorithms.apply_restrictions.**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.

```
expr = MultiFunction.reuse_if_untouched
```

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*)

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

geometric_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.

```
expr = MultiFunction.reuse_if_untouched
```

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

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*)

reference_grad(*expr*, **ops*)

reference_value(*expr*, **ops*)

terminal(*expr*)

ufl.algorithms.balancing.**balance_modified_terminal**(*expr*)

ufl.algorithms.balancing.**balance_modifiers**(*expr*)

ufl.algorithms.change_to_reference module

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

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

cell_avg(*o*, **dummy_ops*)

coefficient_derivative(*o*, **dummy_ops*)

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

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

coefficient_derivative(*o*)

expr(*o*, **ops*)
Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

grad(*o*)

reference_grad(*o*)

terminal(*o*)

ufl.algorithms.change_to_reference.**change_integrand_geometry_representation**(*integrand*, *scale*, *integral_type*)
Change integrand geometry to the right representations.

ufl.algorithms.change_to_reference.**change_to_reference_grad**(*e*)
Change Grad objects in expression to products of JacobianInverse and ReferenceGrad.

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

@param e: An Expr or Form.

ufl.algorithms.check_arities module

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

argument(*o*)

cell_avg(*o*, *a*)

component_tensor(*o*, *a*, *i*)

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*)

terminal(*o*)

variable(*o*, *f*, *l*)

exception ufl.algorithms.check_arities.**ArityMismatch**
Bases: ufl.log.UFLException

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.

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*)

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.

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'

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_int*

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 decremeneted by _reduce_degree.

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*)
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 heuristicaly 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_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.

expr_list(*v*, **o*)

expr_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*)

ufl.algorithms.estimate_degrees.**estimate_total_polynomial_degree**(*e*, *default_degree=1*, *element_replace_map={}*)
Estimate total polynomial degree of integrand.

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

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

ufl.algorithms.expand_compounds module

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

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.**block_split**(*form*, *ix*, *iy=0*)

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 > 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 > 0, providing whatever Argument its operand does.

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

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

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

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

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

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

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

list_tensor(*x*, **ops*)

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

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

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

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

real(*x*, *arg*)
A linear operator with a single operand accepting arity > 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**(*form*, *coefficient*)
Compute the a-norm of a Coefficient given a form a.

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

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

This works simply by replacing the last Argument with a Coefficient on the same function space (element). The form returned will thus have one Argument less and one additional Coefficient at the end if no Coefficient has been provided.

ufl.algorithms.formtransformations.**compute_form_adjoint**(*form*, *reordered_arguments=None*)
Compute the adjoint of a bilinear form.

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

ufl.algorithms.formtransformations.**compute_form_arities**(*form*)
Return set of arities of terms present in form.

ufl.algorithms.formtransformations.**compute_form_functional**(*form*)
Compute the functional part of a form, that is the terms independent of Arguments.

(Used for testing, not sure if it's useful for anything?)

ufl.algorithms.formtransformations.**compute_form_lhs**(*form*)
Compute the left hand side of a form.

Example:

a = u*v*dx + f*v*dx a = lhs(a) -> u*v*dx

ufl.algorithms.formtransformations.**compute_form_rhs**(*form*)
Compute the right hand side of a form.

Example:

a = u*v*dx + f*v*dx L = rhs(a) -> -f*v*dx

ufl.algorithms.formtransformations.**compute_form_with_arity**(*form*, *arity*, *arguments=None*)
Compute parts of form of given arity.

ufl.algorithms.formtransformations.**zero_expr**(*e*)

ufl.algorithms.map_integrands module

Basic algorithms for applying functions to subexpressions.

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

ufl.algorithms.map_integrands.**map_integrands**(*function*, *form*, *only_integral_type=None*)
Apply transform(expression) to each integrand expression in form, or to form if it is an Expr.

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'

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*)
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.

imag(*o*, *a*)

real(*o*, *a*)

terminal(*t*, **ops*)

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.

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

This is a poorly designed algorithm. It is used in some tests, please do not use for anything else.

index(*o*)

multi_index(*o*)

zero(*o*)

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

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.

```
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.

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

variable(*o*)

class ufl.algorithms.transformer.**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*)

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.

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.

expr = MultiFunction.reuse_if_untouched

as a default rule.

reuse_variable(*o*)

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

undefined(*o*)
Trigger error.

visit(*o*)

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

variable(*o*)

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

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

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

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

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

ufl.algorithms.traversal module

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

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

Module contents

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

ufl.algorithms.**estimate_total_polynomial_degree**(*e*, *default_degree=1*, *element_replace_map={}*)
Estimate total polynomial degree of integrand.

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

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

ufl.algorithms.**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.**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.**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.

expr = MultiFunction.reuse_if_untouched

as a default rule.

reuse_variable(*o*)

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

undefined(*o*)
Trigger error.

visit(*o*)

class ufl.algorithms.**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.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

undefined(*o*, **args*)
Trigger error for types with missing handlers.

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

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.

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

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

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

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.

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

Example:

a = u*v*dx + f*v*dx a = lhs(a) -> u*v*dx

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

Example:

a = u*v*dx + f*v*dx L = rhs(a) -> -f*v*dx

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?)

ufl.algorithms.**compute_form_signature**(*form*, *renumbering*)

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

ufl.finiteelement package

Submodules

ufl.finiteelement.brokenelement module

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.

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.

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.**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*)
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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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:

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

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

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.

ufl_free_indices

ufl_index_dimensions

ufl_shape = ()

class ufl.algebra.**Product**(*a*, *b*)
Bases: ufl.core.operator.Operator

The product of two or more UFL objects.

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

ufl_free_indices

ufl_index_dimensions

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.

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.algebra.**Sum**(*a*, *b*)
Bases: ufl.core.operator.Operator

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

ufl_free_indices

ufl_index_dimensions

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 corresponding 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 corresponding to the subelements.

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

ufl.argument.**TrialFunctions**(*function_space*)
UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

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.

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.averaging.**FacetAvg**(*f*)
Bases: ufl.core.operator.Operator

evaluate(*x*, *mapping*, *component*, *index_values*)
Performs an approximate symbolic evaluation, since we dont have a cell.

ufl_free_indices

ufl_index_dimensions

ufl_shape

1.3.7 ufl.cell module

Types for representing a cell.

class ufl.cell.**AbstractCell**(*topological_dimension*, *geometric_dimension*)
Bases: object

Representation of an abstract finite element cell with only the dimensions known.

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.**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.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*, *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.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.

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.

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.

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).

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 = dx/dXf = dx/dX dX/dXf = J * 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 = dX_i/dXf_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.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**(*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'

class ufl.classes.**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.classes.**JacobianInverse**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The inverse of the Jacobian.

Represents the inverse of a square Jacobian or the pseudo-inverse of a non-square Jacobian.

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

name = 'K'

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

class ufl.classes.**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'

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**(*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.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).

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).

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.

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

UFL geometry representation: The volume of the reference cell.

name = 'reference_cell_volume'

class ufl.classes.**ReferenceFacetVolume**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

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

name = 'reference_facet_volume'

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

UFL geometry representation: The volume of the cell.

name = 'volume'

class ufl.classes.**Circumradius**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The circumradius of the cell.

name = 'circumradius'

class ufl.classes.**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.classes.**FacetArea**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The area of the facet.

name = 'facetarea'

class ufl.classes.**MinCellEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The minimum edge length of the cell.

name = 'mincelledgelength'

class ufl.classes.**MaxCellEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The maximum edge length of the cell.

name = 'maxcelledgelength'

class ufl.classes.**MinFacetEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The minimum edge length of the facet.

name = 'minfacetedgelength'

class ufl.classes.**MaxFacetEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The maximum edge length of the facet.

name = 'maxfacetedgelength'

class ufl.classes.**CellOrientation**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The orientation (+1/-1) of the current cell.

For non-manifold cells (tdim == gdim), this equals the sign of the Jacobian determinant, i.e. +1 if the physical cell is oriented the same way as the reference cell and -1 otherwise.

For manifold cells of tdim==gdim-1 this is input data belonging to the mesh, used to distinguish between the sides of the manifold.

name = 'cell_orientation'

class ufl.classes.**FacetOrientation**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The orientation (+1/-1) of the current facet relative to the reference cell.

name = 'facet_orientation'

class ufl.classes.**QuadratureWeight**(*domain*)
Bases: *ufl.geometry.GeometricQuantity*

UFL geometry representation: The current quadrature weight.

Only used inside a quadrature context.

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

name = 'weight'

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

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

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()

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*

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.classes.**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.classes.**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 = ()

class ufl.classes.**ListTensor**(**expressions*)
Bases: ufl.core.operator.Operator

UFL operator type: Wraps a list of expressions into a tensor valued expression of one higher rank.

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

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.classes.**ComponentTensor**(*expression*, *indices*)
Bases: ufl.core.operator.Operator

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

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

indices()

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.classes.**Argument**(*function_space*, *number*, *part=None*)
Bases: ufl.core.terminal.FormArgument

UFL value: Representation of an argument to a form.

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.**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.classes.**Variable**(*expression*, *label=None*)
Bases: ufl.core.operator.Operator

A Variable is a representative for another expression.

It will be used by the end-user mainly for defining a quantity to differentiate w.r.t. using diff. Example:

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

evaluate(*x*, *mapping*, *component*, *index_values*)
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

class ufl.classes.**Sum**(*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.**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.

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

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**(*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**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Exp**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Ln**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

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

class ufl.classes.**Cos**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Sin**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Tan**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Cosh**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Sinh**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Tanh**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Acos**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Asin**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Atan**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.classes.**Atan2**(*arg1*, *arg2*)
Bases: ufl.core.operator.Operator

evaluate(*x*, *mapping*, *component*, *index_values*)
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*, *mapping*, *component*, *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*, *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.**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*, *mapping*, *component*, *index_values*)
Performs an approximate symbolic evaluation, since we dont have a cell.

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.classes.**FacetAvg**(*f*)
Bases: ufl.core.operator.Operator

evaluate(*x*, *mapping*, *component*, *index_values*)
Performs an approximate symbolic evaluation, since we dont have a cell.

ufl_free_indices

ufl_index_dimensions

ufl_shape

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.

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 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 $c_0 \to c_1$ 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*)
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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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:

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

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

geometric_dimension()
Return the dimension of the space this domain is embedded in.

topological_dimension()
Return the dimension of the topology of this domain.

class ufl.classes.**Mesh**(*coordinate_element*, *ufl_id=None*, *cargo=None*)
Bases: *ufl.domain.AbstractDomain*

Symbolic representation of a mesh.

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**(**function_spaces*)
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.**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> b = a.reconstruct(expand_compounds(a.integrand())) c = a.reconstruct(metadata={'quadrature_degree':2})

subdomain_data()
Return the domain data of this integral.

subdomain_id()
Return the subdomain id of this integral.

ufl_domain()
Return the integration domain of this integral.

class ufl.classes.**Form**(*integrals*)
Bases: object

Description of a weak form consisting of a sum of integrals over subdomains.

arguments()
Return all Argument objects found in form.

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

coefficients()
Return all Coefficient objects found in form.

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_type(*integral_type*)
Return a sequence of all integrals with a particular domain type.

max_subdomain_ids()
Returns a mapping on the form `{domain:{integral_type:max_subdomain_id}}`.

signature()
Signature for use with jit cache (independent of incidental numbering of indices etc.)

subdomain_data()
Returns a mapping on the form `{domain:{integral_type: subdomain_data}}`.

ufl_cell()
Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()
Return the single geometric integration domain occuring in the form.

Fails if multiple domains are found.

NB! This does not include domains of coefficients defined on other meshes, look at form data for that additional information.

ufl_domains()
Return the geometric integration domains occuring 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.

ufl.coefficient.**Constant**(*domain*, *count=None*)
UFL value: Represents a globally constant scalar valued coefficient.

ufl.coefficient.**TensorConstant**(*domain*, *shape=None*, *symmetry=None*, *count=None*)
UFL value: Represents a globally constant tensor valued coefficient.

ufl.coefficient.**VectorConstant**(*domain*, *dim=None*, *count=None*)
UFL value: Represents a globally constant vector valued coefficient.

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.

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*

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.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 = ()

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.

1.3.14 ufl.differentiation module

Differential operators.

class ufl.differentiation.**CoefficientDerivative**(*integrand*, *coefficients*, *arguments*, *coefficient_derivatives*)
Bases: *ufl.differentiation.Derivative*

Derivative of the integrand of a form w.r.t. the degrees of freedom in a discrete Coefficient.

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.differentiation.**CompoundDerivative**(*operands*)
Bases: *ufl.differentiation.Derivative*

Base class for all compound derivative types.

class ufl.differentiation.**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.differentiation.**Curl**(*f*)
Bases: *ufl.differentiation.CompoundDerivative*

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.differentiation.**Derivative**(*operands*)
Bases: ufl.core.operator.Operator

Base class for all derivative types.

class ufl.differentiation.**Div**(*f*)
Bases: *ufl.differentiation.CompoundDerivative*

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.differentiation.**Grad**(*f*)
Bases: *ufl.differentiation.CompoundDerivative*

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

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.differentiation.**NablaDiv**(*f*)
Bases: *ufl.differentiation.CompoundDerivative*

ufl_free_indices

ufl_index_dimensions

ufl_shape

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

1.3.15 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.

1.3.16 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.

1.3.17 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

1.3.18 ufl.exprequals module

ufl.exprequals.**expr_equals**(*self*, *other*)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions are mathematically equal or equivalent! Used by sets and dicts.

ufl.exprequals.**measure_collisions**(*equals_func*)

ufl.exprequals.**nonrecursive_expr_equals**(*self*, *other*)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions are mathematically equal or equivalent! Used by sets and dicts.

ufl.exprequals.**print_collisions**()

ufl.exprequals.**recursive_expr_equals**(*self*, *other*)
Checks whether the two expressions are represented the exact same way. This does not check if the expressions are mathematically equal or equivalent! Used by sets and dicts.

1.3.19 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

1.3.20 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.

`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_type`(*integral_type*)

Return a sequence of all integrals with a particular domain type.

`max_subdomain_ids`()

Returns a mapping on the form `{domain:{integral_type:max_subdomain_id}}`.

`signature`()

Signature for use with jit cache (independent of incidental numbering of indices etc.)

`subdomain_data`()

Returns a mapping on the form `{domain:{integral_type: subdomain_data}}`.

ufl_cell()
Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()
Return the single geometric integration domain occuring in the form.

Fails if multiple domains are found.

NB! This does not include domains of coefficients defined on other meshes, look at form data for that additional information.

ufl_domains()
Return the geometric integration domains occuring 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).

1.3.21 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.**block_split**(*form*, *ix*, *iy=0*)
UFL form operator: Given a linear or bilinear form on a mixed space, extract the block correspoinding to the indices ix, iy.

Example:

a = inner(grad(u), grad(v))*dx + div(u)*q*dx + div(v)*p*dx a = block_split(a, 0, 0) -> inner(grad(u), grad(v))*dx

ufl.formoperators.**derivative**(*form*, *coefficient*, *argument=None*, *coefficient_derivatives=None*)
UFL form operator: Compute the Gateaux derivative of *form* w.r.t. *coefficient* in direction of *argument*.

If the argument is omitted, a new Argument is created in the same space as the coefficient, with argument number one higher than the highest one in the form.

The resulting form has one additional Argument in the same finite element space as the coefficient.

A tuple of 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 form operator: Given a bilinear form *a* and a coefficient *f*, return the functional $a(f, f)$.

`ufl.formoperators.`**`functional`**(*form*)
UFL form operator: Extract the functional part of form.

`ufl.formoperators.`**`lhs`**(*form*)
UFL form operator: Given a combined bilinear and linear form, extract the left hand side (bilinear form part).

Example:

```
a = u*v*dx + f*v*dx
a = lhs(a) -> u*v*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 = u*v*dx + f*v*dx
L = rhs(a) -> -f*v*dx
```

`ufl.formoperators.`**`sensitivity_rhs`**(*a*, *u*, *L*, *v*)
UFL form operator: Compute the right hand side for a sensitivity calculation system.

The derivation behind this computation is as follows. Assume *a*, *L* to be bilinear and linear forms corresponding to the assembled linear system

$$Ax = b.$$

Where *x* is the vector of the discrete function corresponding to *u*. Let *v* be some scalar variable this equation depends on. Then we can write

$$0 = \frac{d}{dv}(Ax - b) = \frac{dA}{dv}x + A\frac{dx}{dv} - \frac{db}{dv},$$
$$A\frac{dx}{dv} = \frac{db}{dv} - \frac{dA}{dv}x,$$

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 $\frac{db}{dv}$ and $\frac{dA}{dv}$ from the forms

```
da = diff(a, v)
dL = diff(L, v)
```

and the action of `da` on `u` by

```
dau = 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 = diff(L, v) - action(diff(a, v), u)
```

or, using this function,

```
dL = sensitivity_rhs(a, u, L, v)
```

ufl.formoperators.**set_list_item**(*li*, *i*, *v*)

ufl.formoperators.**system**(*form*)
UFL form operator: Split a form into the left hand side and right hand side, see `lhs` and `rhs`.

ufl.formoperators.**zero_lists**(*shape*)

1.3.22 ufl.functionspace module

Types for representing function spaces.

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

ufl_sub_spaces()

1.3.23 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 = dX_i/dXf_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.

FJ_ij = dx_i/dXf_j

The FacetJacobian is the product of the Jacobian and CellFacetJacobian:

FJ = dx/dXf = dx/dX dX/dXf = J * 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.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'

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'

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.**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.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.

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.geometry.**MaxCellEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The maximum edge length of the cell.

name = 'maxcelledgelength'

class ufl.geometry.**MaxFacetEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The maximum edge length of the facet.

name = 'maxfacetedgelength'

class ufl.geometry.**MinCellEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The minimum edge length of the cell.

name = 'mincelledgelength'

class ufl.geometry.**MinFacetEdgeLength**(*domain*)
Bases: *ufl.geometry.GeometricFacetQuantity*

UFL geometry representation: The minimum edge length of the facet.

name = 'minfacetedgelength'

class ufl.geometry.**QuadratureWeight**(*domain*)
Bases: *ufl.geometry.GeometricQuantity*

UFL geometry representation: The current quadrature weight.

Only used inside a quadrature context.

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

name = 'weight'

class ufl.geometry.**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.geometry.**ReferenceCellVolume**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The volume of the reference cell.

name = '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).

1.3.24 ufl.index_combination_utils module

Utilities for analysing and manipulating free index tuples

ufl.index_combination_utils.**create_slice_indices**(*component*, *shape*, *fi*)

ufl.index_combination_utils.**merge_nonoverlapping_indices**(*a*, *b*)
Merge non-overlapping free indices into one representation.

Example: C[i,j,r,s] = outer(A[i,s], B[j,r]) A, B -> (i,j,r,s), (idim,jdim,rdim,sdim)

ufl.index_combination_utils.**merge_overlapping_indices**(*afi*, *afid*, *bfi*, *bfid*)
Merge overlapping free indices into one free and one repeated representation.

Example: C[j,r] := A[i,j,k] * B[i,r,k] A, B -> (j,r), (jdim,rdim), (i,k), (idim,kdim)

ufl.index_combination_utils.**merge_unique_indices**(*afi*, *afid*, *bfi*, *bfid*)
Merge two pairs of (index ids, index dimensions) sequences into one pair without duplicates.

The id tuples afi, bfi are assumed already sorted by id. Given a list of (id, dim) tuples already sorted by id, return a unique list with duplicates removed. Also checks that the dimensions of duplicates are matching.

ufl.index_combination_utils.**remove_indices**(*fi*, *fid*, *rfi*)

ufl.index_combination_utils.**unique_sorted_indices**(*indices*)
Given a list of (id, dim) tuples already sorted by id, return a unique list with duplicates removed. Also checks that the dimensions of duplicates are matching.

1.3.25 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 = ()

1.3.26 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

1.3.27 ufl.integral module

The Integral class.

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.

1.3.28 ufl.log module

This module provides functions used by the UFL implementation to output messages. These may be redirected by the user of UFL.

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.

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.

1.3.29 ufl.mathfunctions module

This module provides basic mathematical functions.

class ufl.mathfunctions.**Acos**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Asin**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Atan**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Atan2**(*arg1*, *arg2*)
Bases: ufl.core.operator.Operator

evaluate(*x*, *mapping*, *component*, *index_values*)
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.

ufl_free_indices = ()

ufl_index_dimensions = ()

ufl_shape = ()

class ufl.mathfunctions.**Sin**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Sinh**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Sqrt**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Tan**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

class ufl.mathfunctions.**Tanh**(*argument*)
Bases: *ufl.mathfunctions.MathFunction*

1.3.30 ufl.measure module

The Measure class.

class ufl.measure.**Measure**(*integral_type*, *domain=None*, *subdomain_id='everywhere'*, *metadata=None*, *subdomain_data=None*)
Bases: object

integral_type()
Return the domain type.

Valid domain types are "cell", "exterior_facet", "interior_facet", etc.

metadata()
Return the integral metadata. This data is not interpreted by UFL. It is passed to the form compiler which can ignore it or use it to compile each integral of a form in a different way.

reconstruct(*integral_type=None*, *subdomain_id=None*, *domain=None*, *metadata=None*, *subdomain_data=None*)
Construct a new Measure object with some properties replaced with new values.

Example: <dm = Measure instance> b = dm.reconstruct(subdomain_id=2) c = dm.reconstruct(metadata={ "quadrature_degree": 3 })

Used by the call operator, so this is equivalent: b = dm(2) c = dm(0, { "quadrature_degree": 3 })

subdomain_data()
Return the integral subdomain_data. This data is not interpreted by UFL. Its intension is to give a context in which the domain id is interpreted.

subdomain_id()
Return the domain id of this measure (integer).

ufl_domain()
Return the domain associated with this measure.

This may be None or a Domain object.

class ufl.measure.**MeasureProduct**(**measures*)
Bases: object

Represents a product of measures.

This is a notational intermediate object to handle the notation

f*(dm1*dm2)

This is work in progress and not functional. It needs support in other parts of ufl and the rest of the code generation chain.

sub_measures()
Return submeasures.

class ufl.measure.**MeasureSum**(**measures*)
Bases: object

Represents a sum of measures.

This is a notational intermediate object to translate the notation

f*(ds(1)+ds(3))

into

f*ds(1) + f*ds(3)

ufl.measure.**as_integral_type**(*integral_type*)
Map short name to long name and require a valid one.

ufl.measure.**integral_types**()
Return a tuple of all domain type strings.

ufl.measure.**measure_names**()
Return a tuple of all measure name strings.

ufl.measure.**register_integral_type**(*integral_type*, *measure_name*)

1.3.31 ufl.objects module

Utility objects for pretty syntax in user code.

1.3.32 ufl.operators module

This module extends the form language with free function operators, which are either already available as member functions on UFL objects or defined as compound operators involving basic operations on the UFL objects.

ufl.operators.**And**(*left*, *right*)
UFL operator: A boolean expression (left and right) for use with conditional.

ufl.operators.**Dn**(*f*)
UFL operator: Take the directional derivative of *f* in the facet normal direction, Dn(f) := dot(grad(f), n).

ufl.operators.**Dt**(*f*)
UFL operator: <Not implemented yet!> The partial derivative of *f* with respect to time.

ufl.operators.**Dx**(*f*, **i*)
UFL operator: Take the partial derivative of *f* with respect to spatial variable number *i*. Equivalent to f.dx(*i).

ufl.operators.**Max**(*x*, *y*)
UFL operator: Take the maximum of *x* and *y*.

ufl.operators.**Min**(*x*, *y*)
UFL operator: Take the minimum of *x* and *y*.

ufl.operators.**Not**(*condition*)
UFL operator: A boolean expression (not condition) for use with conditional.

ufl.operators.**Or**(*left*, *right*)
UFL operator: A boolean expression (left or right) for use with `conditional`.

ufl.operators.**acos**(*f*)
UFL operator: Take the inverse cosine of *f*.

ufl.operators.**asin**(*f*)
UFL operator: Take the inverse 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

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

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[:].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_div(v) = v[i].dx(i)

nabla_div(T)[:] = T[i,:].dx(i)

for vector expressions v, and arbitrary rank tensor expressions T.

See also: *div()*

ufl.operators.**nabla_grad**(*f*)
UFL operator: Take the gradient of *f*.

This operator follows the grad convention where

nabla_grad(s)[i] = s.dx(i)

nabla_grad(v)[i,j] = v[j].dx(i)

nabla_grad(T)[i,:] = T[:].dx(i)

for scalar expressions s, vector expressions v, and arbitrary rank tensor expressions T.

See also: *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()`.

1.3.33 ufl.permutation module

This module provides utility functions for computing permutations and generating index lists.

ufl.permutation.**build_component_numbering**(*shape*, *symmetry*)
Build a numbering of components within the given value shape, taking into consideration a symmetry mapping which leaves the mapping noncontiguous. Returns a dict { component -> numbering } and an ordered list of components [numbering -> component]. The dict contains all components while the list only contains the ones not mapped by the symmetry mapping.

ufl.permutation.**compute_indices**(*shape*)
Compute all index combinations for given shape

ufl.permutation.**compute_indices2**(*shape*)
Compute all index combinations for given shape

ufl.permutation.**compute_order_tuples**(*k*, *n*)
Compute all tuples of n integers such that the sum is k

ufl.permutation.**compute_permutation_pairs**(*j*, *k*)
Compute all permutations of j + k elements from (0, j + k) in rising order within (0, j) and (j, j + k) respectively.

ufl.permutation.**compute_permutations**(*k*, *n*, *skip=None*)
Compute all permutations of k elements from (0, n) in rising order. Any elements that are contained in the list skip are not included.

ufl.permutation.**compute_sign**(*permutation*)
Compute sign by sorting.

1.3.34 ufl.precedence module

Precedence handling.

ufl.precedence.**assign_precedences**(*precedence_list*)
Given a precedence list, assign ints to class._precedence.

ufl.precedence.**build_precedence_list**()

ufl.precedence.**build_precedence_mapping**(*precedence_list*)
Given a precedence list, build a dict with class->int mappings. Utility function used by some external code.

ufl.precedence.**parstr**(*child*, *parent*, *pre='('*, *post=')'*, *format=<class 'str'>*)

1.3.35 ufl.protocols module

ufl.protocols.**id_or_none**(*obj*)
Returns None if the object is None, obj.ufl_id() if available, or id(obj) if not.

This allows external libraries to implement an alternative to id(obj) in the ufl_id() function, such that ufl can identify objects as the same without knowing about their types.

ufl.protocols.**metadata_equal**(*a*, *b*)

ufl.protocols.**metadata_hashdata**(*md*)

1.3.36 ufl.referencevalue module

Representation of the reference value of a function.

class ufl.referencevalue.**ReferenceValue**(*f*)
Bases: ufl.core.operator.Operator

Representation of the reference cell value of a form argument.

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

ufl_free_indices = ()

ufl_index_dimensions = ()

ufl_shape

1.3.37 ufl.restriction module

Restriction operations.

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

1.3.38 ufl.sobolevspace module

This module defines a symbolic heirarchy 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 differerent 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.

1.3.39 ufl.sorting module

This module contains a sorting rule for expr objects that is more robust w.r.t. argument numbering than using repr.

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*)

1.3.40 ufl.split_functions module

Algorithm for splitting a Coefficient or Argument into subfunctions.

ufl.split_functions.**split**(*v*)
UFL operator: If v is a Coefficient or Argument in a mixed space, returns a tuple with the function components corresponding to the subelements.

1.3.41 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 = ()

ufl_index_dimensions = ()

ufl_shape = ()

class ufl.tensoralgebra.**Deviatoric**(*A*)
Bases: *ufl.tensoralgebra.CompoundTensorOperator*

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.tensoralgebra.**Dot**(*a*, *b*)
Bases: *ufl.tensoralgebra.CompoundTensorOperator*

ufl_free_indices

ufl_index_dimensions

ufl_shape

class ufl.tensoralgebra.**Inner**(*a*, *b*)
Bases: *ufl.tensoralgebra.CompoundTensorOperator*

ufl_free_indices

ufl_index_dimensions

ufl_shape = ()

class ufl.tensoralgebra.**Inverse**(*A*)
Bases: *ufl.tensoralgebra.CompoundTensorOperator*

ufl_free_indices = ()

ufl_index_dimensions = ()

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*

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

1.3.42 ufl.tensors module

Classes used to group scalar expressions into expressions with rank > 0.

class ufl.tensors.**ComponentTensor**(*expression*, *indices*)
Bases: ufl.core.operator.Operator

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

evaluate(*x*, *mapping*, *component*, *index_values*)
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:

```
(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]*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*)

1.3.43 ufl.variable module

Defines the Variable and Label classes, used to label expressions as variables for differentiation.

class ufl.variable.**Label**(*count=None*)
Bases: `ufl.core.terminal.Terminal`

count()

is_cellwise_constant()

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:

```
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

1.3.44 Module contents

The Unified Form Language is an embedded domain specific language for definition of variational forms intended for finite element discretization. More precisely, it defines a fixed interface for choosing finite element spaces and defining expressions for weak forms in a notation close to the mathematical one.

This Python module contains the language as well as algorithms to work with it.

- To import the language, type:

```
from ufl import *
```

- To import the underlying classes an UFL expression tree is built from, type

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

- Various algorithms for working with UFL expression trees can be accessed by

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

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
- HCurl
```

- Elements:

```
- FiniteElement
- MixedElement
- VectorElement
- TensorElement
- EnrichedElement
- NodalEnrichedElement
- RestrictedElement
- TensorProductElement
- HDivElement
- HCurlElement
- BrokenElement
- FacetElement
- InteriorElement
```

- Function spaces:

```
- FunctionSpace
```

- Arguments:

```
- Argument
- TestFunction
- TrialFunction
- Arguments
- TestFunctions
- TrialFunctions
```

- Coefficients:

```
- Coefficient
- Constant
- VectorConstant
- TensorConstant
```

- Splitting form arguments in mixed spaces:

```
- split
```

- Literal constants:

```
- Identity
- PermutationSymbol
```

- Geometric quantities:

```
- SpatialCoordinate
- FacetNormal
- CellNormal
- CellVolume
- 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
```

(continues on next page)

(continued from previous page)

```
- 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).

class ufl.**Jacobian**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

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

$$J_{ij} = \frac{dx_i}{dX_j}$$

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.**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.**JacobianInverse**(*domain*)
Bases: *ufl.geometry.GeometricCellQuantity*

UFL geometry representation: The inverse of the Jacobian.

Represents the inverse of a square Jacobian or the pseudo-inverse of a non-square Jacobian.

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

name = 'K'

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

class ufl.**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 $c_0 \to c_1$ 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**(*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*)
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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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 $c_0 \to c_1$ 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:

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

Given bases $\{\phi_{j_i}\}$ of the spaces V_i for $i = 1, ..., d$, $\{\phi_{j_1} \otimes \phi_{j_2} \otimes \cdots \otimes \phi_{j_d}\}$ 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.**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.**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 corresponding to the subelements.

ufl.**TestFunctions**(*function_space*)
UFL value: Create a TestFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

ufl.**TrialFunctions**(*function_space*)
UFL value: Create a TrialFunction in a mixed space, and return a tuple with the function components corresponding to the subelements.

class ufl.**Coefficient**(*function_space*, *count=None*)
Bases: ufl.core.terminal.FormArgument

UFL form argument type: Representation of a form coefficient.

count()

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.

ufl.**Constant**(*domain*, *count=None*)
UFL value: Represents a globally constant scalar valued coefficient.

ufl.**VectorConstant**(*domain*, *dim=None*, *count=None*)
UFL value: Represents a globally constant vector valued coefficient.

ufl.**TensorConstant**(*domain*, *shape=None*, *symmetry=None*, *count=None*)
UFL value: Represents a globally constant tensor valued coefficient.

ufl.**split**(*v*)
UFL operator: If v is a Coefficient or Argument in a mixed space, returns a tuple with the function components corresponding to the subelements.

class ufl.**PermutationSymbol**(*dim*)
Bases: *ufl.constantvalue.ConstantValue*

UFL literal type: Representation of a permutation symbol.

This is also known as the Levi-Civita symbol, antisymmetric symbol, or alternating symbol.

evaluate(*x*, *mapping*, *component*, *index_values*)
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

ufl.**zero**(**shape*)
UFL literal constant: Return a zero tensor with the given shape.

ufl.**as_ufl**(*expression*)
Converts expression to an Expr if possible.

class ufl.**Index**(*count=None*)
Bases: ufl.core.multiindex.IndexBase

UFL value: An index with no value assigned.

Used to represent free indices in Einstein indexing notation.

count()

ufl.**indices**(*n*)
UFL value: Return a tuple of n new Index objects.

ufl.**as_tensor**(*expressions*, *indices=None*)
UFL operator: Make a tensor valued expression.

This works in two different ways, by using indices or lists.

1) Returns A such that A [*indices*] = *expressions*. If *indices* are provided, *expressions* must be a scalar valued expression with all the provided indices among its free indices. This operator will then map each of these indices to a tensor axis, thereby making a tensor valued expression from a scalar valued expression with free indices.

2) Returns A such that $A[k, ...]$ = *expressions*[k]. If no indices are provided, *expressions* must be a list or tuple of expressions. The expressions can also consist of recursively nested lists to build higher rank tensors.

ufl.**as_vector**(*expressions*, *index=None*)
UFL operator: As as_tensor(), but limited to rank 1 tensors.

ufl.**as_matrix**(*expressions*, *indices=None*)
UFL operator: As *as_tensor()*, but limited to rank 2 tensors.

ufl.**relabel**(*A*, *indexmap*)
UFL operator: Relabel free indices of A with new indices, using the given mapping.

ufl.**unit_vector**(*i*, *d*)
UFL value: A constant unit vector in direction *i* with dimension *d*.

ufl.**unit_vectors**(*d*)
UFL value: A tuple of constant unit vectors in all directions with dimension *d*.

ufl.**unit_matrix**(*i*, *j*, *d*)
UFL value: A constant unit matrix in direction *i*,*j* with dimension *d*.

ufl.**unit_matrices**(*d*)
UFL value: A tuple of constant unit matrices in all directions with dimension *d*.

ufl.**rank**(*f*)
UFL operator: The rank of *f*.

ufl.**shape**(*f*)
UFL operator: The shape of *f*.

ufl.**conj**(*f*)
UFL operator: The complex conjugate of *f*

ufl.**real**(*f*)
UFL operator: The real part of *f*

ufl.**imag**(*f*)
UFL operator: The imaginary part of *f*

ufl.**outer**(**operands*)
UFL operator: Take the outer product of two or more operands. The complex conjugate of the first argument is taken.

ufl.**inner**(*a*, *b*)
UFL operator: Take the inner product of *a* and *b*. The complex conjugate of the second argument is taken.

ufl.**dot**(*a*, *b*)
UFL operator: Take the dot product of *a* and *b*. The complex conjugate of the second argument is taken.

ufl.**cross**(*a*, *b*)
UFL operator: Take the cross product of *a* and *b*.

ufl.**perp**(*v*)
UFL operator: Take the perp of *v*, i.e. $(-v_1, +v_0)$.

ufl.**det**(*A*)
UFL operator: Take the determinant of *A*.

ufl.**inv**(*A*)
UFL operator: Take the inverse of *A*.

ufl.**cofac**(*A*)
UFL operator: Take the cofactor of *A*.

ufl.**transpose**(*A*)
UFL operator: Take the transposed of tensor A.

ufl.**tr**(*A*)
UFL operator: Take the trace of *A*.

ufl.**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.**diag_vector**(*A*)
UFL operator: Take the diagonal part of rank 2 tensor *A* and return as a vector.

See also `diag`.

ufl.**dev**(*A*)
UFL operator: Take the deviatoric part of *A*.

ufl.**skew**(*A*)
UFL operator: Take the skew symmetric part of *A*.

ufl.**sym**(*A*)
UFL operator: Take the symmetric part of *A*.

ufl.**sqrt**(*f*)
UFL operator: Take the square root of *f*.

ufl.**exp**(*f*)
UFL operator: Take the exponential of *f*.

ufl.**ln**(*f*)
UFL operator: Take the natural logarithm of *f*.

ufl.**erf**(*f*)
UFL operator: Take the error function of *f*.

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.**Or**(*left*, *right*)
UFL operator: A boolean expression (left or right) for use with `conditional`.

ufl.**Not**(*condition*)
UFL operator: A boolean expression (not condition) for use with `conditional`.

ufl.**conditional**(*condition*, *true_value*, *false_value*)
UFL operator: A conditional expression, taking the value of *true_value* when *condition* evaluates to `true` and *false_value* otherwise.

ufl.**sign**(*x*)
UFL operator: Take the sign (+1 or -1) of *x*.

ufl.**max_value**(*x*, *y*)
UFL operator: Take the maximum of *x* and *y*.

ufl.**min_value**(*x*, *y*)
UFL operator: Take the minimum of *x* and *y*.

ufl.**Max**(*x*, *y*)
UFL operator: Take the maximum of *x* and *y*.

ufl.**Min**(*x*, *y*)
UFL operator: Take the minimum of *x* and *y*.

ufl.**variable**(*e*)
UFL operator: Define a variable representing the given expression, see also `diff()`.

ufl.**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.**Dx**(*f*, **i*)
UFL operator: Take the partial derivative of *f* with respect to spatial variable number *i*. Equivalent to `f.dx(*i)`.

ufl.**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()*

ufl.**div**(*f*)
UFL operator: Take the divergence of *f*.

This operator follows the div convention where

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

div(T)[:] = T[:,i].dx(i)

for vector expressions v, and arbitrary rank tensor expressions T.

See also: *nabla_div()*

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_grad(s)[i] = s.dx(i)

nabla_grad(v)[i,j] = v[j].dx(i)

nabla_grad(T)[i,:] = T[:].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_div(v) = v[i].dx(i)

nabla_div(T)[:] = 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, Dn(f) := dot(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.

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_type(*integral_type*)
Return a sequence of all integrals with a particular domain type.

max_subdomain_ids()
Returns a mapping on the form {domain:{integral_type:max_subdomain_id}}.

signature()
Signature for use with jit cache (independent of incidental numbering of indices etc.)

subdomain_data()
Returns a mapping on the form {domain:{integral_type: subdomain_data}}.

ufl_cell()
Return the single cell this form is defined on, fails if multiple cells are found.

ufl_domain()
Return the single geometric integration domain occuring in the form.

Fails if multiple domains are found.

NB! This does not include domains of coefficients defined on other meshes, look at form data for that additional information.

ufl_domains()
Return the geometric integration domains occuring 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.**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.

class ufl.**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.

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*, *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:

```
a = u*v*dx + f*v*dx
L = rhs(a) -> -f*v*dx
```

ufl.**lhs**(*form*)
UFL form operator: Given a combined bilinear and linear form, extract the left hand side (bilinear form part).

Example:

```
a = u*v*dx + f*v*dx
a = lhs(a) -> u*v*dx
```

ufl.**block_split**(*form*, *ix*, *iy=0*)
UFL form operator: Given a linear or bilinear form on a mixed space, extract the block correspoinding to the indices ix, iy.

Example:

a = inner(grad(u), grad(v))*dx + div(u)*q*dx + div(v)*p*dx a = block_split(a, 0, 0) -> inner(grad(u), grad(v))*dx

ufl.**system**(*form*)
UFL form operator: Split a form into the left hand side and right hand side, see `lhs` and `rhs`.

ufl.**functional**(*form*)
UFL form operator: Extract the functional part of form.

ufl.**adjoint**(*form*, *reordered_arguments=None*)
UFL form operator: Given a combined bilinear form, compute the adjoint form by changing the ordering (count) of the test and trial functions, 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

```
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 $\frac{db}{dv}$ and $\frac{dA}{dv}$ from the forms

```
da = diff(a, v)
dL = diff(L, v)
```

and the action of `da` on `u` by

```
dau = 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 = diff(L, v) - action(diff(a, v), u)
```

or, using this function,

```
dL = sensitivity_rhs(a, u, L, v)
```

1.4 Release notes

1.4.1 Changes in the next release

Summary of changes

- Add support for complex valued elements

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 quantites `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 d622c74 (22 March 2010). A warning was introduced in fcbc5ff (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`
- Outer product element support in degree estimation
- Add `TraceElement`, `InteriorElement`, `FacetElement`, `BrokenElement`
- Add `OuterProductCell` to valid `Real` elements
- Add `_cache` member to form for use by external frameworks
- Add Sobolev space `HEin`
- Add measures `dI`, `dO`, `dC` for interface, overlap, cutcell
- Remove `Measure` constants
- Remove `cell2D` and `cell3D`
- Implement `reference_value` in `apply_restrictions`
- Rename point integral to vertex integral and kept `*dP` syntax
- Replace lambda functions in `ufl_type` with named functions for nicer stack traces
- Minor bugfixes, removal of unused code and cleanups

[FIXME: These links don't belong here, should go under API reference somehow.]

- genindex
- modindex

Python Module Index

u

ufl, 159
ufl.algebra, 83
ufl.algorithms, 73
ufl.algorithms.ad, 46
ufl.algorithms.analysis, 46
ufl.algorithms.apply_algebra_lowering, 47
ufl.algorithms.apply_derivatives, 48
ufl.algorithms.apply_function_pullbacks, 53
ufl.algorithms.apply_geometry_lowering, 53
ufl.algorithms.apply_integral_scaling, 54
ufl.algorithms.apply_restrictions, 54
ufl.algorithms.balancing, 56
ufl.algorithms.change_to_reference, 57
ufl.algorithms.check_arities, 58
ufl.algorithms.check_restrictions, 59
ufl.algorithms.checks, 59
ufl.algorithms.comparison_checker, 59
ufl.algorithms.compute_form_data, 60
ufl.algorithms.coordinate_derivative_helpers, 60
ufl.algorithms.domain_analysis, 61
ufl.algorithms.elementtransformations, 62
ufl.algorithms.estimate_degrees, 62
ufl.algorithms.expand_compounds, 65
ufl.algorithms.expand_indices, 65
ufl.algorithms.formdata, 66
ufl.algorithms.formfiles, 66
ufl.algorithms.formsplitter, 67
ufl.algorithms.formtransformations, 67
ufl.algorithms.map_integrands, 69
ufl.algorithms.multifunction, 70
ufl.algorithms.remove_complex_nodes, 70
ufl.algorithms.renumbering, 70
ufl.algorithms.replace, 70
ufl.algorithms.signature, 71
ufl.algorithms.transformer, 71
ufl.algorithms.traversal, 73
ufl.argument, 85
ufl.assertions, 86
ufl.averaging, 86
ufl.cell, 86
ufl.checks, 88
ufl.classes, 88
ufl.coefficient, 121
ufl.compound_expressions, 122
ufl.conditional, 123
ufl.constantvalue, 124
ufl.differentiation, 126
ufl.domain, 128
ufl.equation, 129
ufl.exprcontainers, 130
ufl.exprequals, 130
ufl.exproperators, 131
ufl.finiteelement, 83
ufl.finiteelement.brokenelement, 77
ufl.finiteelement.elementlist, 77
ufl.finiteelement.enrichedelement, 77
ufl.finiteelement.facetelement, 78
ufl.finiteelement.finiteelement, 78
ufl.finiteelement.finiteelementbase, 79
ufl.finiteelement.hdivcurl, 80
ufl.finiteelement.interiorelement, 81
ufl.finiteelement.mixedelement, 81
ufl.finiteelement.restrictedelement, 82
ufl.finiteelement.tensorproductelement, 83
ufl.form, 131
ufl.formoperators, 132
ufl.functionspace, 134
ufl.geometry, 135
ufl.index_combination_utils, 142
ufl.indexed, 143
ufl.indexsum, 143

ufl.integral, 143
ufl.log, 144
ufl.mathfunctions, 145
ufl.measure, 147
ufl.objects, 148
ufl.operators, 148
ufl.permutation, 153
ufl.precedence, 153
ufl.protocols, 153
ufl.referencevalue, 154
ufl.restriction, 154
ufl.sobolevspace, 154
ufl.sorting, 155
ufl.split_functions, 155
ufl.tensoralgebra, 155
ufl.tensors, 157
ufl.variable, 158

Index

A

Abs (class in ufl.algebra), 83
Abs (class in ufl.classes), 102
abs() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
abs() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
abs() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
AbstractCell (class in ufl), 163
AbstractCell (class in ufl.cell), 86
AbstractCell (class in ufl.classes), 111
AbstractDomain (class in ufl), 164
AbstractDomain (class in ufl.classes), 117
AbstractDomain (class in ufl.domain), 128
AbstractFunctionSpace (class in ufl.classes), 118
AbstractFunctionSpace (class in ufl.functionspace), 134
accumulate_integrands_with_same_metadata() (in module ufl.algorithms.domain_analysis), 61
Acos (class in ufl.classes), 110
Acos (class in ufl.mathfunctions), 145
acos() (in module ufl), 176
acos() (in module ufl.operators), 149
acos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
action() (in module ufl), 181
action() (in module ufl.formoperators), 132
add_indent() (ufl.log.Logger method), 144
add_logfile() (ufl.log.Logger method), 144
adj_expr() (in module ufl.compound_expressions), 122
adj_expr_2x2() (in module ufl.compound_expressions), 122
adj_expr_3x3() (in module ufl.compound_expressions), 122
adj_expr_4x4() (in module ufl.compound_expressions), 122
adjoint() (in module ufl), 182
adjoint() (in module ufl.formoperators), 132
affine_mesh() (in module ufl.domain), 129
altenative_dot() (ufl.algorithms.apply_algebra_lowering.LowerCompoundA method), 47
alternative_inner() (ufl.algorithms.apply_algebra_lowering.LowerCompoun method), 47
always_reconstruct() (ufl.algorithms.Transformer method), 74
always_reconstruct() (ufl.algorithms.transformer.Transformer method), 72
analyse_key() (in module ufl.exproperators), 131
And() (in module ufl), 177
And() (in module ufl.operators), 148
AndCondition (class in ufl.classes), 108
AndCondition (class in ufl.conditional), 123
apply_algebra_lowering() (in module ufl.algorithms.apply_algebra_lowering), 47
apply_coordinate_derivatives() (in module ufl.algorithms.apply_derivatives), 52
apply_default_restrictions() (in module ufl.algorithms.apply_restrictions), 56
apply_derivatives() (in module ufl.algorithms.apply_derivatives), 52
apply_function_pullbacks() (in module ufl.algorithms.apply_function_pullbacks), 53
apply_geometry_lowering() (in module ufl.algorithms.apply_geometry_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), 73
apply_transformer() (in module ufl.algorithms.transformer), 72
Argument (class in ufl), 172
Argument (class in ufl.argument), 85

Argument (class in ufl.classes), 100
argument() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
argument() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
argument() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
argument() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
argument() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
argument() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
argument() (ufl.algorithms.check_arities.ArityChecker method), 58
argument() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
argument() (ufl.algorithms.FormSplitter method), 76
argument() (ufl.algorithms.formsplitter.FormSplitter method), 67
argument() (ufl.algorithms.formtransformations.PartExtracter method), 67
argument() (ufl.classes.ComplexValue method), 99
argument() (ufl.constantvalue.ComplexValue method), 125
Arguments() (in module ufl), 173
Arguments() (in module ufl.argument), 85
arguments() (ufl.classes.Form method), 120
arguments() (ufl.Form method), 179
arguments() (ufl.form.Form method), 131
ArityChecker (class in ufl.algorithms.check_arities), 58
ArityMismatch, 58
as_cell() (in module ufl), 163
as_cell() (in module ufl.cell), 87
as_domain() (in module ufl), 164
as_domain() (in module ufl.domain), 129
as_form() (in module ufl.form), 132
as_integral_type() (in module ufl.measure), 148
as_matrix() (in module ufl), 174
as_matrix() (in module ufl.tensors), 157
as_scalar() (in module ufl.tensors), 157
as_scalars() (in module ufl.tensors), 157
as_tensor() (in module ufl), 174
as_tensor() (in module ufl.tensors), 157
as_ufl() (in module ufl), 174
as_ufl() (in module ufl.constantvalue), 126
as_vector() (in module ufl), 174
as_vector() (in module ufl.tensors), 158
Asin (class in ufl.classes), 110
Asin (class in ufl.mathfunctions), 145
asin() (in module ufl), 176
asin() (in module ufl.operators), 149
asin() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
assign_precedences() (in module ufl.precedence), 153
Atan (class in ufl.classes), 110
Atan (class in ufl.mathfunctions), 145
atan() (in module ufl), 176
atan() (in module ufl.operators), 149
atan() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 49
Atan2 (class in ufl.classes), 110
Atan2 (class in ufl.mathfunctions), 145
atan_2() (in module ufl), 176
atan_2() (in module ufl.operators), 149
atan_2() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
atan_2() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 62
attach_coordinate_derivatives() (in module ufl.algorithms.coordinate_derivative_helpers), 61
attach_estimated_degrees() (in module ufl.algorithms.compute_form_data), 60
avg() (in module ufl), 178
avg() (in module ufl.operators), 149

B

balance_modified_terminal() (in module ufl.algorithms.balancing), 56
balance_modifiers() (in module ufl.algorithms.balancing), 56
BalanceModifiers (class in ufl.algorithms.balancing), 56
begin() (ufl.log.Logger method), 144
bessel_function() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
bessel_I() (in module ufl), 176
bessel_I() (in module ufl.operators), 149
bessel_i() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
bessel_J() (in module ufl), 176
bessel_J() (in module ufl.operators), 149
bessel_j() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
bessel_K() (in module ufl), 176
bessel_K() (in module ufl.operators), 149
bessel_k() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
bessel_Y() (in module ufl), 176
bessel_Y() (in module ufl.operators), 149
bessel_y() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
BesselFunction (class in ufl.classes), 110
BesselFunction (class in ufl.mathfunctions), 146
BesselI (class in ufl.classes), 110
BesselI (class in ufl.mathfunctions), 146
BesselJ (class in ufl.classes), 110
BesselJ (class in ufl.mathfunctions), 146

BesselK (class in ufl.classes), 110
BesselK (class in ufl.mathfunctions), 146
BesselY (class in ufl.classes), 110
BesselY (class in ufl.mathfunctions), 146
binary_condition() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
BinaryCondition (class in ufl.classes), 107
BinaryCondition (class in ufl.conditional), 123
block_split() (in module ufl), 181
block_split() (in module ufl.algorithms.formsplitter), 67
block_split() (in module ufl.formoperators), 132
BrokenElement (class in ufl), 171
BrokenElement (class in ufl.classes), 117
BrokenElement (class in ufl.finiteelement.brokenelement), 77
build_component_numbering() (in module ufl.permutation), 153
build_integral_data() (in module ufl.algorithms.domain_analysis), 61
build_precedence_list() (in module ufl.precedence), 153
build_precedence_mapping() (in module ufl.precedence), 153

C

canonical_element_description() (in module ufl.finiteelement.elementlist), 77
Cell (class in ufl), 163
Cell (class in ufl.cell), 87
Cell (class in ufl.classes), 111
cell() (ufl.classes.FiniteElementBase method), 112
cell() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
cell() (ufl.FiniteElementBase method), 167
cell_avg() (in module ufl), 178
cell_avg() (in module ufl.operators), 149
cell_avg() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
cell_avg() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
cell_avg() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
cell_avg() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
cell_avg() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
cell_avg() (ufl.algorithms.balancing.BalanceModifiers method), 56
cell_avg() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
cell_avg() (ufl.algorithms.check_arities.ArityChecker method), 58
cell_avg() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
cell_avg() (ufl.algorithms.formtransformations.PartExtracter method), 67
cell_coordinate() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
cell_coordinate() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
cell_coordinate() (ufl.algorithms.apply_geometry_lowering.GeometryLowe method), 53
cell_coordinate() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
cell_diameter() (ufl.algorithms.apply_geometry_lowering.GeometryLoweri method), 53
cell_normal() (ufl.algorithms.apply_geometry_lowering.GeometryLowering method), 53
cell_volume() (ufl.algorithms.apply_geometry_lowering.GeometryLowering method), 53
CellAvg (class in ufl.averaging), 86
CellAvg (class in ufl.classes), 110
CellCoordinate (class in ufl.classes), 90
CellCoordinate (class in ufl.geometry), 135
CellDiameter (class in ufl), 165
CellDiameter (class in ufl.classes), 96
CellDiameter (class in ufl.geometry), 135
CellEdgeVectors (class in ufl.classes), 93
CellEdgeVectors (class in ufl.geometry), 135
CellFacetJacobian (class in ufl.classes), 92
CellFacetJacobian (class in ufl.geometry), 135
CellFacetJacobianDeterminant (class in ufl.classes), 94
CellFacetJacobianDeterminant (class in ufl.geometry), 136
CellFacetJacobianInverse (class in ufl.classes), 95
CellFacetJacobianInverse (class in ufl.geometry), 136
CellFacetOrigin (class in ufl.classes), 91
CellFacetOrigin (class in ufl.geometry), 136
cellname() (ufl.Cell method), 163
cellname() (ufl.cell.Cell method), 87
cellname() (ufl.cell.TensorProductCell method), 87
cellname() (ufl.classes.Cell method), 111
cellname() (ufl.classes.TensorProductCell method), 112
cellname() (ufl.TensorProductCell method), 164
CellNormal (class in ufl), 166
CellNormal (class in ufl.classes), 95
CellNormal (class in ufl.geometry), 136
CellOrientation (class in ufl.classes), 97
CellOrientation (class in ufl.geometry), 136
CellOrigin (class in ufl.classes), 91
CellOrigin (class in ufl.geometry), 137
CellVertices (class in ufl.classes), 93
CellVertices (class in ufl.geometry), 137
CellVolume (class in ufl), 165
CellVolume (class in ufl.classes), 96
CellVolume (class in ufl.geometry), 137
change_integrand_geometry_representation() (in module ufl.algorithms.change_to_reference), 57

change_to_reference_grad() (in module ufl.algorithms), 75
change_to_reference_grad() (in module ufl.algorithms.change_to_reference), 57
check_form_arity() (in module ufl.algorithms.check_arities), 58
check_integrand_arity() (in module ufl.algorithms.check_arities), 58
check_restrictions() (in module ufl.algorithms.check_restrictions), 59
CheckComparisons (class in ufl.algorithms.comparison_checker), 59
Circumradius (class in ufl), 165
Circumradius (class in ufl.classes), 96
Circumradius (class in ufl.geometry), 137
circumradius() (ufl.algorithms.apply_geometry_lowering.GeometryLowering method), 53
cmp_expr() (in module ufl.sorting), 155
codeterminant_expr_nxn() (in module ufl.compound_expressions), 122
Coefficient (class in ufl), 173
Coefficient (class in ufl.classes), 100
Coefficient (class in ufl.coefficient), 121
coefficient() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
coefficient() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
coefficient() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
coefficient() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
coefficient() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
coefficient() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
coefficient() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
coefficient_derivative() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
coefficient_derivative() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
coefficient_derivative() (ufl.algorithms.change_to_reference.ChangeToReferenceGrad method), 57
coefficient_derivative() (ufl.algorithms.change_to_reference.ChangeToReferenceGrad method), 57
coefficient_derivative() (ufl.algorithms.replace.Replacer method), 70
coefficient_numbering() (ufl.classes.Form method), 120
coefficient_numbering() (ufl.Form method), 179
coefficient_numbering() (ufl.form.Form method), 131
CoefficientDerivative (class in ufl.classes), 106
CoefficientDerivative (class in ufl.differentiation), 126
Coefficients() (in module ufl), 173
Coefficients() (in module ufl.coefficient), 122
coefficients() (ufl.classes.Form method), 120
coefficients() (ufl.Form method), 179
coefficients() (ufl.form.Form method), 131
cofac() (in module ufl), 175
cofac() (in module ufl.operators), 149
Cofactor (class in ufl.classes), 104
Cofactor (class in ufl.tensoralgebra), 155
cofactor() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cofactor() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
cofactor_expr() (in module ufl.compound_expressions), 122
cofactor_expr_2x2() (in module ufl.compound_expressions), 122
cofactor_expr_3x3() (in module ufl.compound_expressions), 122
cofactor_expr_4x4() (in module ufl.compound_expressions), 122
compare() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
ComplexComparisonError, 60
ComplexNodeRemoval (class in ufl.algorithms.remove_complex_nodes), 70
ComplexValue (class in ufl.classes), 98
ComplexValue (class in ufl.constantvalue), 124
component() (ufl.algorithms.expand_indices.IndexExpander method), 65
component_tensor() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
component_tensor() (ufl.algorithms.check_arities.ArityChecker method), 58
component_tensor() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
component_tensor() (ufl.algorithms.expand_indices.IndexExpander method), 65
component_tensor() (ufl.algorithms.formtransformations.PartExtracter method), 67
ComponentTensor (class in ufl.classes), 100
ComponentTensor (class in ufl.tensors), 157
NEW() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
OldComponentTensor() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
CompoundDerivative (class in ufl.classes), 106
CompoundDerivative (class in ufl.differentiation), 126
CompoundTensorOperator (class in ufl.classes), 103
CompoundTensorOperator (class in ufl.tensoralgebra), 155
compute_energy_norm() (in module ufl.algorithms), 76
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
compute_form_action() (in module ufl.algorithms), 76
compute_form_action() (in module ufl.algorithms.formtransformations), 69
compute_form_adjoint() (in module ufl.algorithms), 76
compute_form_adjoint() (in module ufl.algorithms.formtransformations), 69
compute_form_arities() (in module ufl.algorithms.formtransformations), 69
compute_form_data() (in module ufl.algorithms), 73
compute_form_data() (in module ufl.algorithms.compute_form_data), 60
compute_form_functional() (in module ufl.algorithms), 76
compute_form_functional() (in module ufl.algorithms.formtransformations), 69
compute_form_lhs() (in module ufl.algorithms), 76
compute_form_lhs() (in module ufl.algorithms.formtransformations), 69
compute_form_rhs() (in module ufl.algorithms), 76
compute_form_rhs() (in module ufl.algorithms.formtransformations), 69
compute_form_signature() (in module ufl.algorithms), 76
compute_form_signature() (in module ufl.algorithms.signature), 71
compute_form_with_arity() (in module ufl.algorithms.formtransformations), 69
compute_indices() (in module ufl.permutation), 153
compute_indices2() (in module ufl.permutation), 153
compute_integrand_scaling_factor() (in module ufl.algorithms.apply_integral_scaling), 54
compute_multiindex_hashdata() (in module ufl.algorithms.signature), 71
compute_order_tuples() (in module ufl.permutation), 153
compute_permutation_pairs() (in module ufl.permutation), 153
compute_permutations() (in module ufl.permutation), 153
compute_sign() (in module ufl.permutation), 153
compute_terminal_hashdata() (in module ufl.algorithms.signature), 71
Condition (class in ufl.classes), 107
Condition (class in ufl.conditional), 123
condition() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
Conditional (class in ufl.classes), 108
Conditional (class in ufl.conditional), 123
conditional() (in module ufl), 177
conditional() (in module ufl.operators), 149
conditional() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
conditional() (ufl.algorithms.check_arities.ArityChecker method), 58
conditional() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
conditional() (ufl.algorithms.expand_indices.IndexExpander method), 66
Conj (class in ufl.algebra), 84
Conj (class in ufl.classes), 102
conj() (in module ufl), 175
conj() (in module ufl.operators), 149
conj() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
conj() (ufl.algorithms.check_arities.ArityChecker method), 58
conj() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
conj() (ufl.algorithms.formtransformations.PartExtracter method), 67
conj() (ufl.algorithms.remove_complex_nodes.ComplexNodeRemoval method), 70
conjugate() (in module ufl.operators), 149
Constant() (in module ufl), 173
Constant() (in module ufl.coefficient), 122
constant_value() (ufl.algorithms.apply_derivatives.GenericDerivativeRulese method), 50
constant_value() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
constant_value() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
ConstantValue (class in ufl.classes), 98
ConstantValue (class in ufl.constantvalue), 125
contraction() (in module ufl.operators), 149
coordinate_derivative() (ufl.algorithms.apply_derivatives.CoordinateDerivat method), 48
coordinate_derivative() (ufl.algorithms.apply_derivatives.DerivativeRuleDis method), 49
coordinate_derivative() (ufl.algorithms.apply_derivatives.GateauxDerivative method), 49
coordinate_derivative() (ufl.algorithms.coordinate_derivative_helpers.Coord method), 60
coordinate_derivative() (ufl.algorithms.estimate_degrees.SumDegreeEstima method), 63
CoordinateDerivative (class in ufl.classes), 106
CoordinateDerivative (class in ufl.differentiation), 127
CoordinateDerivativeIsOutermostChecker (class in ufl.algorithms.coordinate_derivative_helpers), 60
CoordinateDerivativeRuleDispatcher (class in ufl.algorithms.apply_derivatives), 48
CoordinateDerivativeRuleset (class in ufl.algorithms.apply_derivatives), 48
CopyTransformer (class in ufl.algorithms.transformer), 71
Cos (class in ufl.classes), 109
Cos (class in ufl.mathfunctions), 146

cos() (in module ufl), 176
cos() (in module ufl.operators), 149
cos() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
Cosh (class in ufl.classes), 109
Cosh (class in ufl.mathfunctions), 146
cosh() (in module ufl), 176
cosh() (in module ufl.operators), 149
cosh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
count() (ufl.classes.Coefficient method), 100
count() (ufl.classes.Index method), 119
count() (ufl.classes.Label method), 101
count() (ufl.classes.SpatialCoordinate method), 90
count() (ufl.Coefficient method), 173
count() (ufl.coefficient.Coefficient method), 121
count() (ufl.geometry.SpatialCoordinate method), 142
count() (ufl.Index method), 174
count() (ufl.SpatialCoordinate method), 165
count() (ufl.variable.Label method), 158
create_nested_lists() (in module ufl.algorithms.apply_function_pullbacks), 53
create_slice_indices() (in module ufl.index_combination_utils), 142
Cross (class in ufl.classes), 104
Cross (class in ufl.tensoralgebra), 155
cross() (in module ufl), 175
cross() (in module ufl.operators), 149
cross() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
cross() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
cross_expr() (in module ufl.compound_expressions), 122
Curl (class in ufl.classes), 107
Curl (class in ufl.differentiation), 127
curl() (in module ufl), 178
curl() (in module ufl.operators), 149
curl() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
curl() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63

D

debug() (ufl.log.Logger method), 144
default_domain() (in module ufl.domain), 129
DefaultRestrictionApplier (class in ufl.algorithms.apply_restrictions), 54
degree() (ufl.classes.FiniteElementBase method), 112
degree() (ufl.classes.MixedElement method), 113
degree() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
degree() (ufl.finiteelement.mixedelement.MixedElement method), 81
degree() (ufl.FiniteElementBase method), 167
degree() (ufl.MixedElement method), 168
deprecate() (ufl.log.Logger method), 144
Derivative (class in ufl.classes), 106
Derivative (class in ufl.differentiation), 127
derivative() (in module ufl), 181
derivative() (in module ufl.formoperators), 132
derivative() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleDispatcher method), 48
derivative() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
derivative() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
derivative() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 54
derivative() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
DerivativeRuleDispatcher (class in ufl.algorithms.apply_derivatives), 48
det() (in module ufl), 175
det() (in module ufl.operators), 149
Determinant (class in ufl.classes), 104
Determinant (class in ufl.tensoralgebra), 155
determinant() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
determinant() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
determinant_expr() (in module ufl.compound_expressions), 122
determinant_expr_2x2() (in module ufl.compound_expressions), 122
determinant_expr_3x3() (in module ufl.compound_expressions), 122
dev() (in module ufl), 175
dev() (in module ufl.operators), 149
Deviatoric (class in ufl.classes), 104
Deviatoric (class in ufl.tensoralgebra), 155
deviatoric() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
deviatoric() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
deviatoric_expr() (in module ufl.compound_expressions), 122
deviatoric_expr_2x2() (in module ufl.compound_expressions), 122
deviatoric_expr_3x3() (in module ufl.compound_expressions), 122
diag() (in module ufl), 175
diag() (in module ufl.operators), 150
diag_vector() (in module ufl), 175
diag_vector() (in module ufl.operators), 150
dicts_lt() (in module ufl.algorithms.domain_analysis), 61
diff() (in module ufl), 177
diff() (in module ufl.operators), 150

dimension() (ufl.classes.IndexSum method), 105
dimension() (ufl.indexsum.IndexSum method), 143
DirectionalSobolevSpace (class in ufl.sobolevspace), 154
Div (class in ufl.classes), 107
Div (class in ufl.differentiation), 127
div() (in module ufl), 177
div() (in module ufl.operators), 150
div() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
div() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
Division (class in ufl.algebra), 84
Division (class in ufl.classes), 102
division() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
division() (ufl.algorithms.check_arities.ArityChecker method), 58
division() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
division() (ufl.algorithms.expand_indices.IndexExpander method), 66
division() (ufl.algorithms.formtransformations.PartExtracter method), 67
Dn() (in module ufl), 178
Dn() (in module ufl.operators), 148
do_comparison_check() (in module ufl.algorithms.comparison_checker), 60
domain (ufl.algorithms.domain_analysis.IntegralData attribute), 61
domain_numbering() (ufl.classes.Form method), 120
domain_numbering() (ufl.Form method), 179
domain_numbering() (ufl.form.Form method), 131
Dot (class in ufl.classes), 103
Dot (class in ufl.tensoralgebra), 155
dot() (in module ufl), 175
dot() (in module ufl.operators), 150
dot() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
dot() (ufl.algorithms.check_arities.ArityChecker method), 58
dot() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
dot() (ufl.algorithms.formtransformations.PartExtracter method), 67
Dt() (in module ufl.operators), 148
Dx() (in module ufl), 177
Dx() (in module ufl.operators), 148
dx() (ufl.classes.Expr method), 89
dyad() (in module ufl.tensors), 158

E

elem_div() (in module ufl), 179
elem_div() (in module ufl.operators), 150
elem_mult() (in module ufl), 178
elem_mult() (in module ufl.operators), 150
elem_op() (in module ufl), 179
elem_op() (in module ufl.operators), 150
elem_op_items() (in module ufl.operators), 150
elem_pow() (in module ufl), 179
elem_pow() (in module ufl.operators), 150
empty() (ufl.classes.Form method), 120
empty() (ufl.Form method), 179
empty() (ufl.form.Form method), 131
enabled_coefficients (ufl.algorithms.domain_analysis.IntegralData attribute), 61
end() (ufl.log.Logger method), 144
energy_norm() (in module ufl), 181
energy_norm() (in module ufl.formoperators), 133
EnrichedElement (class in ufl), 169
EnrichedElement (class in ufl.classes), 115
EnrichedElement (class in ufl.finiteelement.enrichedelement), 77
EnrichedElementBase (class in ufl.finiteelement.enrichedelement), 78
EQ (class in ufl.classes), 107
EQ (class in ufl.conditional), 123
eq() (in module ufl), 176
eq() (in module ufl.operators), 150
equals() (ufl.classes.Form method), 120
equals() (ufl.Form method), 179
equals() (ufl.form.Form method), 131
Equation (class in ufl.classes), 121
Equation (class in ufl.equation), 129
Erf (class in ufl.classes), 110
Erf (class in ufl.mathfunctions), 146
erf() (in module ufl), 176
erf() (in module ufl.operators), 150
erf() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
error() (ufl.log.Logger method), 144
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), 83
evaluate() (ufl.algebra.Conj method), 84
evaluate() (ufl.algebra.Division method), 84
evaluate() (ufl.algebra.Imag method), 84
evaluate() (ufl.algebra.Power method), 84
evaluate() (ufl.algebra.Product method), 84
evaluate() (ufl.algebra.Real method), 84
evaluate() (ufl.algebra.Sum method), 85
evaluate() (ufl.averaging.CellAvg method), 86
evaluate() (ufl.averaging.FacetAvg method), 86
evaluate() (ufl.classes.Abs method), 102
evaluate() (ufl.classes.AndCondition method), 108
evaluate() (ufl.classes.Atan2 method), 110
evaluate() (ufl.classes.BesselFunction method), 110

evaluate() (ufl.classes.CellAvg method), 110
evaluate() (ufl.classes.ComponentTensor method), 100
evaluate() (ufl.classes.Conditional method), 108
evaluate() (ufl.classes.Conj method), 102
evaluate() (ufl.classes.Division method), 102
evaluate() (ufl.classes.EQ method), 108
evaluate() (ufl.classes.Erf method), 110
evaluate() (ufl.classes.Expr method), 89
evaluate() (ufl.classes.FacetAvg method), 111
evaluate() (ufl.classes.GE method), 108
evaluate() (ufl.classes.Grad method), 106
evaluate() (ufl.classes.GT method), 108
evaluate() (ufl.classes.Identity method), 99
evaluate() (ufl.classes.Imag method), 103
evaluate() (ufl.classes.Indexed method), 99
evaluate() (ufl.classes.IndexSum method), 105
evaluate() (ufl.classes.LE method), 108
evaluate() (ufl.classes.ListTensor method), 99
evaluate() (ufl.classes.Ln method), 109
evaluate() (ufl.classes.LT method), 108
evaluate() (ufl.classes.MathFunction method), 109
evaluate() (ufl.classes.MaxValue method), 109
evaluate() (ufl.classes.MinValue method), 109
evaluate() (ufl.classes.MultiIndex method), 97
evaluate() (ufl.classes.NE method), 108
evaluate() (ufl.classes.NotCondition method), 108
evaluate() (ufl.classes.OrCondition method), 108
evaluate() (ufl.classes.PermutationSymbol method), 99
evaluate() (ufl.classes.Power method), 102
evaluate() (ufl.classes.Product method), 102
evaluate() (ufl.classes.Real method), 103
evaluate() (ufl.classes.ReferenceGrad method), 106
evaluate() (ufl.classes.ReferenceValue method), 111
evaluate() (ufl.classes.Restricted method), 105
evaluate() (ufl.classes.ScalarValue method), 98
evaluate() (ufl.classes.SpatialCoordinate method), 90
evaluate() (ufl.classes.Sum method), 102
evaluate() (ufl.classes.Terminal method), 89
evaluate() (ufl.classes.Variable method), 101
evaluate() (ufl.classes.Zero method), 98
evaluate() (ufl.conditional.AndCondition method), 123
evaluate() (ufl.conditional.Conditional method), 123
evaluate() (ufl.conditional.EQ method), 123
evaluate() (ufl.conditional.GE method), 123
evaluate() (ufl.conditional.GT method), 123
evaluate() (ufl.conditional.LE method), 124
evaluate() (ufl.conditional.LT method), 124
evaluate() (ufl.conditional.MaxValue method), 124
evaluate() (ufl.conditional.MinValue method), 124
evaluate() (ufl.conditional.NE method), 124
evaluate() (ufl.conditional.NotCondition method), 124
evaluate() (ufl.conditional.OrCondition method), 124
evaluate() (ufl.constantvalue.Identity method), 125
evaluate() (ufl.constantvalue.PermutationSymbol method), 125
evaluate() (ufl.constantvalue.ScalarValue method), 125
evaluate() (ufl.constantvalue.Zero method), 126
evaluate() (ufl.differentiation.Grad method), 127
evaluate() (ufl.differentiation.ReferenceGrad method), 128
evaluate() (ufl.geometry.SpatialCoordinate method), 142
evaluate() (ufl.Identity method), 174
evaluate() (ufl.indexed.Indexed method), 143
evaluate() (ufl.indexsum.IndexSum method), 143
evaluate() (ufl.mathfunctions.Atan2 method), 145
evaluate() (ufl.mathfunctions.BesselFunction method), 146
evaluate() (ufl.mathfunctions.Erf method), 146
evaluate() (ufl.mathfunctions.Ln method), 146
evaluate() (ufl.mathfunctions.MathFunction method), 146
evaluate() (ufl.PermutationSymbol method), 173
evaluate() (ufl.referencevalue.ReferenceValue method), 154
evaluate() (ufl.restriction.Restricted method), 154
evaluate() (ufl.SpatialCoordinate method), 165
evaluate() (ufl.tensors.ComponentTensor method), 157
evaluate() (ufl.tensors.ListTensor method), 157
evaluate() (ufl.variable.Variable method), 159
execute_ufl_code() (in module ufl.algorithms.formfiles), 66
Exp (class in ufl.classes), 109
Exp (class in ufl.mathfunctions), 146
exp() (in module ufl), 176
exp() (in module ufl.operators), 150
exp() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
expand_compounds() (in module ufl.algorithms), 75
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_terminal() (in module ufl.assertions), 86
expecting_true_ufl_scalar() (in module ufl.assertions), 86
Expr (class in ufl.classes), 88
expr() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
expr() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleDispatche method), 48
expr() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49

expr() (ufl.algorithms.apply_derivatives.GenericDerivativeRemover method), 50
expr() (ufl.algorithms.apply_function_pullbacks.FunctionPullbackApplier method), 53
expr() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
expr() (ufl.algorithms.balancing.BalanceModifiers method), 56
expr() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
expr() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 57
expr() (ufl.algorithms.check_arities.ArityChecker method), 58
expr() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
expr() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
expr() (ufl.algorithms.coordinate_derivative_helpers.CoordinateDerivativeIsOutermostChecker method), 60
expr() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 63
expr() (ufl.algorithms.FormSplitter method), 76
expr() (ufl.algorithms.formsplitter.FormSplitter method), 67
expr() (ufl.algorithms.formtransformations.PartExtracter method), 67
expr() (ufl.algorithms.MultiFunction method), 75
expr() (ufl.algorithms.remove_complex_nodes.ComplexNodeRemover method), 70
expr() (ufl.algorithms.replace.Replacer method), 71
expr() (ufl.algorithms.ReuseTransformer method), 73
expr() (ufl.algorithms.Transformer method), 74
expr() (ufl.algorithms.transformer.CopyTransformer method), 71
expr() (ufl.algorithms.transformer.ReuseTransformer method), 71
expr() (ufl.algorithms.transformer.Transformer method), 72
expr_equals() (in module ufl.exprequals), 130
expr_list() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
expr_mapping() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
ExprData (class in ufl.algorithms.formdata), 66
expression() (ufl.classes.Variable method), 101
expression() (ufl.variable.Variable method), 159
ExprList (class in ufl.classes), 105
ExprList (class in ufl.exprcontainers), 130
ExprMapping (class in ufl.classes), 105
ExprMapping (class in ufl.exprcontainers), 130
ExprTupleKey (class in ufl.algorithms.domain_analysis), 61
exterior_derivative() (in module ufl), 178
exterior_derivative() (in module ufl.operators), 150
extract_arguments() (in module ufl.algorithms), 76
extract_arguments() (in module ufl.algorithms.analysis), 46
extract_arguments_and_coefficients() (in module ufl.algorithms.analysis), 46
extract_coefficients() (in module ufl.algorithms), 75
extract_coefficients() (in module ufl.algorithms.analysis), 46
extract_component() (ufl.classes.FiniteElementBase method), 112
extract_component() (ufl.classes.MixedElement method), 113
extract_component() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
extract_component() (ufl.finiteelement.mixedelement.MixedElement method), 81
extract_component() (ufl.FiniteElementBase method), 167
extract_component() (ufl.MixedElement method), 168
extract_domains() (in module ufl.domain), 129
extract_elements() (in module ufl.algorithms), 75
extract_elements() (in module ufl.algorithms.analysis), 46
extract_reference_component() (ufl.classes.FiniteElementBase method), 112
extract_reference_component() (ufl.classes.MixedElement method), 113
extract_reference_component() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
extract_reference_component() (ufl.finiteelement.mixedelement.MixedElement method), 81
extract_reference_component() (ufl.FiniteElementBase method), 167
extract_reference_component() (ufl.MixedElement method), 168
extract_sub_elements() (in module ufl.algorithms), 75
extract_sub_elements() (in module ufl.algorithms.analysis), 46
extract_subelement_component() (ufl.classes.FiniteElementBase method), 112
extract_subelement_component() (ufl.classes.MixedElement method), 114
extract_subelement_component() (ufl.classes.TensorElement method), 114
extract_subelement_component() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
extract_subelement_component() (ufl.finiteelement.mixedelement.MixedElement method), 81

extract_subelement_component() (ufl.finiteelement.mixedelement.TensorElement method), 82
extract_subelement_component() (ufl.FiniteElementBase method), 167
extract_subelement_component() (ufl.MixedElement method), 168
extract_subelement_component() (ufl.TensorElement method), 169
extract_subelement_reference_component() (ufl.classes.FiniteElementBase method), 112
extract_subelement_reference_component() (ufl.classes.MixedElement method), 114
extract_subelement_reference_component() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
extract_subelement_reference_component() (ufl.finiteelement.mixedelement.MixedElement method), 81
extract_subelement_reference_component() (ufl.FiniteElementBase method), 167
extract_subelement_reference_component() (ufl.MixedElement method), 168
extract_type() (in module ufl.algorithms), 75
extract_type() (in module ufl.algorithms.analysis), 46
extract_unique_domain() (in module ufl.domain), 129
extract_unique_elements() (in module ufl.algorithms), 75
extract_unique_elements() (in module ufl.algorithms.analysis), 46

F

facet_area() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 53
facet_area() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 54
facet_avg() (in module ufl), 178
facet_avg() (in module ufl.operators), 150
facet_avg() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
facet_avg() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
facet_avg() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
facet_avg() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
facet_avg() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
facet_avg() (ufl.algorithms.balancing.BalanceModifiers method), 56
facet_avg() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
facet_avg() (ufl.algorithms.check_arities.ArityChecker method), 58
facet_avg() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
facet_avg() (ufl.algorithms.formtransformations.PartExtracter method), 67
facet_cell_coordinate() (ufl.algorithms.apply_geometry_lowering.GeometryLowerer method), 54
facet_coordinate() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
facet_jacobian() (ufl.algorithms.apply_geometry_lowering.GeometryLowerer method), 54
facet_jacobian() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 54
facet_jacobian_determinant() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
facet_jacobian_determinant() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 54
facet_jacobian_inverse() (ufl.algorithms.apply_geometry_lowering.GeometryLowerer method), 54
facet_jacobian_inverse() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
facet_normal() (ufl.algorithms.apply_geometry_lowering.GeometryLowerer method), 54
facet_normal() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
facet_normal() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
facet_origin() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
FacetArea (class in ufl), 166
FacetArea (class in ufl.classes), 96
FacetArea (class in ufl.geometry), 137
FacetAvg (class in ufl.averaging), 86
FacetAvg (class in ufl.classes), 111
FacetCoordinate (class in ufl.classes), 90
FacetCoordinate (class in ufl.geometry), 137
FacetEdgeVectors (class in ufl.classes), 94
FacetEdgeVectors (class in ufl.geometry), 138
FacetElement() (in module ufl), 172
FacetElement() (in module ufl.classes), 117
FacetElement() (in module ufl.finiteelement.facetelement), 78
FacetJacobian (class in ufl.classes), 92
FacetJacobian (class in ufl.geometry), 138
FacetJacobianDeterminant (class in ufl.classes), 94
FacetJacobianDeterminant (class in ufl.geometry), 138
FacetJacobianInverse (class in ufl.classes), 95
FacetJacobianInverse (class in ufl.geometry), 139
FacetNormal (class in ufl), 166
FacetNormal (class in ufl.classes), 95
FacetNormal (class in ufl.geometry), 139
FacetOrientation (class in ufl.classes), 97
FacetOrientation (class in ufl.geometry), 139

FacetOrigin (class in ufl.classes), 91
FacetOrigin (class in ufl.geometry), 139
family() (ufl.classes.FiniteElementBase method), 112
family() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
family() (ufl.FiniteElementBase method), 167
feec_element() (in module ufl.finiteelement.elementlist), 77
FileData (class in ufl.algorithms.formfiles), 66
find_geometric_dimension() (in module ufl.domain), 129
FiniteElement (class in ufl), 168
FiniteElement (class in ufl.classes), 113
FiniteElement (class in ufl.finiteelement.finiteelement), 78
FiniteElementBase (class in ufl), 167
FiniteElementBase (class in ufl.classes), 112
FiniteElementBase (class in ufl.finiteelement.finiteelementbase), 79
FixedIndex (class in ufl.classes), 118
fixme() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
flattened_sub_element_mapping() (ufl.classes.TensorElement method), 114
flattened_sub_element_mapping() (ufl.finiteelement.mixedelement.TensorElement method), 82
flattened_sub_element_mapping() (ufl.TensorElement method), 169
FloatValue (class in ufl.classes), 99
FloatValue (class in ufl.constantvalue), 125
Form (class in ufl), 179
Form (class in ufl.classes), 120
Form (class in ufl.form), 131
form_argument() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
form_argument() (ufl.algorithms.apply_function_pullbacks.FunctionPullbackApplier method), 53
form_argument() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
form_argument() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
form_argument() (ufl.algorithms.expand_indices.IndexExpander method), 66
FormArgument (class in ufl.classes), 89
format_float() (in module ufl.constantvalue), 126
FormData (class in ufl.algorithms.formdata), 66
FormSplitter (class in ufl.algorithms), 76
FormSplitter (class in ufl.algorithms.formsplitter), 67
free_indices() (ufl.classes.ExprList method), 105
free_indices() (ufl.classes.ExprMapping method), 105
free_indices() (ufl.exprcontainers.ExprList method), 130
free_indices() (ufl.exprcontainers.ExprMapping method), 130
from_numpy_to_lists() (in module ufl.tensors), 158
functional() (in module ufl), 182
functional() (in module ufl.formoperators), 133
FunctionPullbackApplier (class in ufl.algorithms.apply_function_pullbacks), 53
FunctionSpace (class in ufl), 172
FunctionSpace (class in ufl.classes), 118
FunctionSpace (class in ufl.functionspace), 134

G

GateauxDerivativeRuleset (class in ufl.algorithms.apply_derivatives), 49
GE (class in ufl.classes), 108
GE (class in ufl.conditional), 123
ge() (in module ufl), 176
ge() (in module ufl.operators), 151
ge() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
generic_pseudo_determinant_expr() (in module ufl.compound_expressions), 122
generic_pseudo_inverse_expr() (in module ufl.compound_expressions), 122
GenericDerivativeRuleset (class in ufl.algorithms.apply_derivatives), 49
geometric_cell_quantity() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
geometric_dimension() (ufl.AbstractCell method), 163
geometric_dimension() (ufl.AbstractDomain method), 164
geometric_dimension() (ufl.cell.AbstractCell method), 86
geometric_dimension() (ufl.classes.AbstractCell method), 111
geometric_dimension() (ufl.classes.AbstractDomain method), 117
geometric_dimension() (ufl.classes.Expr method), 89
geometric_dimension() (ufl.classes.Form method), 121
geometric_dimension() (ufl.domain.AbstractDomain method), 128
geometric_dimension() (ufl.Form method), 179
geometric_dimension() (ufl.form.Form method), 131
geometric_facet_quantity() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
geometric_quantity() (ufl.algorithms.apply_derivatives.CoordinateDerivativ method), 48
geometric_quantity() (ufl.algorithms.apply_derivatives.GateauxDerivativeR method), 49
geometric_quantity() (ufl.algorithms.apply_derivatives.GenericDerivativeRu method), 50
geometric_quantity() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
geometric_quantity() (ufl.algorithms.apply_derivatives.ReferenceGradRules method), 52

geometric_quantity() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
geometric_quantity() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
geometric_quantity() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
GeometricCellQuantity (class in ufl.classes), 90
GeometricCellQuantity (class in ufl.geometry), 139
GeometricFacetQuantity (class in ufl.classes), 90
GeometricFacetQuantity (class in ufl.geometry), 139
GeometricQuantity (class in ufl.classes), 90
GeometricQuantity (class in ufl.geometry), 140
GeometryLoweringApplier (class in ufl.algorithms.apply_geometry_lowering), 53
get_handler() (ufl.log.Logger method), 144
get_logfile_handler() (ufl.log.Logger method), 144
get_logger() (ufl.log.Logger method), 144
Grad (class in ufl.classes), 106
Grad (class in ufl.differentiation), 127
grad() (in module ufl), 177
grad() (in module ufl.operators), 151
grad() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleDispatcher method), 48
grad() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
grad() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
grad() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
grad() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
grad() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
grad() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
grad() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
grad() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
grad() (ufl.algorithms.balancing.BalanceModifiers method), 56
grad() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
grad() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 57
grad() (ufl.algorithms.check_arities.ArityChecker method), 58
grad() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
grad() (ufl.algorithms.expand_indices.IndexExpander method), 66
grad() (ufl.algorithms.formtransformations.PartExtracter method), 68
GradRuleset (class in ufl.algorithms.apply_derivatives), 51
group_integrals_by_domain_and_type() (in module ufl.algorithms.domain_analysis), 61
group_integrals_by_domain_and_type() (in module ufl.algorithms.domain_analysis), 62
GT (class in ufl.classes), 108
GT (class in ufl.conditional), 123
gt() (in module ufl), 177
gt() (in module ufl.operators), 151
gt() (ufl.algorithms.comparison_checker.CheckComparisons method), 59

H

has_exact_type() (in module ufl.algorithms.analysis), 46
has_simplex_facets() (ufl.AbstractCell method), 163
has_simplex_facets() (ufl.Cell method), 163
has_simplex_facets() (ufl.cell.AbstractCell method), 87
has_simplex_facets() (ufl.cell.Cell method), 87
has_simplex_facets() (ufl.cell.TensorProductCell method), 87
has_simplex_facets() (ufl.classes.AbstractCell method), 111
has_simplex_facets() (ufl.classes.Cell method), 111
has_simplex_facets() (ufl.classes.TensorProductCell method), 112
has_simplex_facets() (ufl.TensorProductCell method), 164
has_type() (in module ufl.algorithms.analysis), 46
HCurlElement (class in ufl), 171
HCurlElement (class in ufl.classes), 116
HCurlElement (class in ufl.finiteelement.hdivcurl), 80
HDivElement (class in ufl), 171
HDivElement (class in ufl.classes), 116
HDivElement (class in ufl.finiteelement.hdivcurl), 80
hypercube() (in module ufl.cell), 88

I

id_or_none() (in module ufl.protocols), 153
Identity (class in ufl), 174
Identity (class in ufl.classes), 99
Identity (class in ufl.constantvalue), 125
Imag (class in ufl.algebra), 84
Imag (class in ufl.classes), 103
imag() (in module ufl), 175
imag() (in module ufl.operators), 151
imag() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
imag() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
imag() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
imag() (ufl.algorithms.formtransformations.PartExtracter method), 68

imag() (ufl.algorithms.remove_complex_nodes.ComplexNodeRemover method), 70
imag() (ufl.classes.ScalarValue method), 98
imag() (ufl.constantvalue.ScalarValue method), 126
increase_order() (in module ufl.algorithms.elementtransformations), 62
independent_operator() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
independent_terminal() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
Index (class in ufl), 174
Index (class in ufl.classes), 118
index() (ufl.algorithms.renumbering.IndexRenumberingTransformer method), 70
index() (ufl.classes.IndexSum method), 105
index() (ufl.indexsum.IndexSum method), 143
index_dimensions() (ufl.classes.ExprList method), 105
index_dimensions() (ufl.classes.ExprMapping method), 105
index_dimensions() (ufl.exprcontainers.ExprList method), 130
index_dimensions() (ufl.exprcontainers.ExprMapping method), 130
index_sum() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
index_sum() (ufl.algorithms.check_arities.ArityChecker method), 58
index_sum() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
index_sum() (ufl.algorithms.expand_indices.IndexExpander method), 66
index_sum() (ufl.algorithms.formtransformations.PartExtracter method), 68
IndexBase (class in ufl.classes), 118
Indexed (class in ufl.classes), 99
Indexed (class in ufl.indexed), 143
indexed() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
indexed() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
indexed() (ufl.algorithms.check_arities.ArityChecker method), 58
indexed() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
indexed() (ufl.algorithms.expand_indices.IndexExpander method), 66
indexed() (ufl.algorithms.formtransformations.PartExtracter method), 68
IndexExpander (class in ufl.algorithms.expand_indices), 65
IndexRenumberingTransformer (class in ufl.algorithms.renumbering), 70
IndexSum (class in ufl.classes), 105
IndexSum (class in ufl.indexsum), 143
indices() (in module ufl), 174
indices() (ufl.classes.ComponentTensor method), 100
indices() (ufl.classes.MultiIndex method), 97
indices() (ufl.tensors.ComponentTensor method), 157
info() (ufl.log.Logger method), 144
info_blue() (ufl.log.Logger method), 145
info_green() (ufl.log.Logger method), 145
info_red() (ufl.log.Logger method), 145
InnerDerivative (class in ufl.classes), 103
Inner (class in ufl.tensoralgebra), 156
inner() (in module ufl), 175
inner() (in module ufl.operators), 151
inner() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
inner() (ufl.algorithms.check_arities.ArityChecker method), 58
inner() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
inner() (ufl.algorithms.formtransformations.PartExtracter method), 68
Integral (class in ufl), 180
Integral (class in ufl.classes), 120
Integral (class in ufl.integral), 143
integral_coefficients (ufl.algorithms.domain_analysis.IntegralData attribute), 61
integral_subdomain_ids() (in module ufl.algorithms.domain_analysis), 62
integral_type (ufl.algorithms.domain_analysis.IntegralData attribute), 61
integral_type() (ufl.classes.Integral method), 120
integral_type() (ufl.classes.Measure method), 119
integral_type() (ufl.Integral method), 180
integral_type() (ufl.integral.Integral method), 143
integral_type() (ufl.Measure method), 180
integral_type() (ufl.measure.Measure method), 147
integral_types() (in module ufl), 181
integral_types() (in module ufl.measure), 148
IntegralData (class in ufl.algorithms.domain_analysis), 61
integrals() (ufl.algorithms.domain_analysis.IntegralData attribute), 61
integrals() (ufl.classes.Form method), 121
integrals() (ufl.Form method), 179
integrals() (ufl.form.Form method), 131
integrals_by_type() (ufl.classes.Form method), 121
integrals_by_type() (ufl.Form method), 179
integrals_by_type() (ufl.form.Form method), 131
integrand() (ufl.classes.Integral method), 120
integrand() (ufl.Integral method), 180
integrand() (ufl.integral.Integral method), 144
InteriorElement() (in module ufl), 172
InteriorElement() (in module ufl.classes), 117
InteriorElement() (in module ufl.finiteelement.interiorelement), 81

interpret_ufl_namespace() (in module ufl.algorithms.formfiles), 66
IntValue (class in ufl.classes), 99
IntValue (class in ufl.constantvalue), 125
inv() (in module ufl), 175
inv() (in module ufl.operators), 151
Inverse (class in ufl.classes), 104
Inverse (class in ufl.tensoralgebra), 156
inverse() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
inverse() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
inverse_expr() (in module ufl.compound_expressions), 122
IrreducibleInt (class in ufl.algorithms.estimate_degrees), 62
is_cellwise_constant() (in module ufl.checks), 88
is_cellwise_constant() (ufl.Argument method), 172
is_cellwise_constant() (ufl.argument.Argument method), 85
is_cellwise_constant() (ufl.classes.Argument method), 100
is_cellwise_constant() (ufl.classes.CellCoordinate method), 90
is_cellwise_constant() (ufl.classes.CellEdgeVectors method), 93
is_cellwise_constant() (ufl.classes.CellFacetJacobian method), 92
is_cellwise_constant() (ufl.classes.CellFacetJacobianDeterminant method), 94
is_cellwise_constant() (ufl.classes.CellFacetJacobianInverse method), 95
is_cellwise_constant() (ufl.classes.CellOrigin method), 91
is_cellwise_constant() (ufl.classes.CellVertices method), 93
is_cellwise_constant() (ufl.classes.Coefficient method), 100
is_cellwise_constant() (ufl.classes.ConstantValue method), 98
is_cellwise_constant() (ufl.classes.EnrichedElement method), 115
is_cellwise_constant() (ufl.classes.FacetCoordinate method), 91
is_cellwise_constant() (ufl.classes.FacetEdgeVectors method), 94
is_cellwise_constant() (ufl.classes.FacetJacobian method), 92
is_cellwise_constant() (ufl.classes.FacetJacobianDeterminant method), 94
is_cellwise_constant() (ufl.classes.FacetJacobianInverse method), 95
is_cellwise_constant() (ufl.classes.FacetNormal method), 95
is_cellwise_constant() (ufl.classes.FiniteElementBase method), 112
is_cellwise_constant() (ufl.classes.GeometricQuantity method), 90
is_cellwise_constant() (ufl.classes.Jacobian method), 92
is_cellwise_constant() (ufl.classes.JacobianDeterminant method), 94
is_cellwise_constant() (ufl.classes.JacobianInverse method), 94
is_cellwise_constant() (ufl.classes.Label method), 101
is_cellwise_constant() (ufl.classes.MixedElement method), 114
is_cellwise_constant() (ufl.classes.MultiIndex method), 97
is_cellwise_constant() (ufl.classes.NodalEnrichedElement method), 115
is_cellwise_constant() (ufl.classes.QuadratureWeight method), 97
is_cellwise_constant() (ufl.classes.ReferenceCellEdgeVectors method), 93
is_cellwise_constant() (ufl.classes.ReferenceFacetEdgeVectors method), 93
is_cellwise_constant() (ufl.classes.RestrictedElement method), 115
is_cellwise_constant() (ufl.classes.SpatialCoordinate method), 90
is_cellwise_constant() (ufl.Coefficient method), 173
is_cellwise_constant() (ufl.coefficient.Coefficient method), 121
is_cellwise_constant() (ufl.constantvalue.ConstantValue method), 125
is_cellwise_constant() (ufl.EnrichedElement method), 170
is_cellwise_constant() (ufl.FacetNormal method), 166
is_cellwise_constant() (ufl.finiteelement.enrichedelement.EnrichedElement method), 78
is_cellwise_constant() (ufl.finiteelement.enrichedelement.NodalEnrichedEle method), 78
is_cellwise_constant() (ufl.finiteelement.finiteelementbase.FiniteElementBa method), 79
is_cellwise_constant() (ufl.finiteelement.mixedelement.MixedElement method), 81
is_cellwise_constant() (ufl.finiteelement.restrictedelement.RestrictedElemer method), 82
is_cellwise_constant() (ufl.FiniteElementBase method), 167
is_cellwise_constant() (ufl.geometry.CellCoordinate method), 135
is_cellwise_constant() (ufl.geometry.CellEdgeVectors method), 135
is_cellwise_constant() (ufl.geometry.CellFacetJacobian method), 135
is_cellwise_constant() (ufl.geometry.CellFacetJacobianDeterminant method), 136

is_cellwise_constant() (ufl.geometry.CellFacetJacobianInverse method), 136
is_cellwise_constant() (ufl.geometry.CellOrigin method), 137
is_cellwise_constant() (ufl.geometry.CellVertices method), 137
is_cellwise_constant() (ufl.geometry.FacetCoordinate method), 138
is_cellwise_constant() (ufl.geometry.FacetEdgeVectors method), 138
is_cellwise_constant() (ufl.geometry.FacetJacobian method), 138
is_cellwise_constant() (ufl.geometry.FacetJacobianDeterminant method), 139
is_cellwise_constant() (ufl.geometry.FacetJacobianInverse method), 139
is_cellwise_constant() (ufl.geometry.FacetNormal method), 139
is_cellwise_constant() (ufl.geometry.GeometricQuantity method), 140
is_cellwise_constant() (ufl.geometry.Jacobian method), 140
is_cellwise_constant() (ufl.geometry.JacobianDeterminant method), 140
is_cellwise_constant() (ufl.geometry.JacobianInverse method), 140
is_cellwise_constant() (ufl.geometry.QuadratureWeight method), 141
is_cellwise_constant() (ufl.geometry.ReferenceCellEdgeVectors method), 141
is_cellwise_constant() (ufl.geometry.ReferenceFacetEdgeVectors method), 141
is_cellwise_constant() (ufl.geometry.SpatialCoordinate method), 142
is_cellwise_constant() (ufl.Jacobian method), 166
is_cellwise_constant() (ufl.JacobianDeterminant method), 167
is_cellwise_constant() (ufl.JacobianInverse method), 167
is_cellwise_constant() (ufl.MixedElement method), 168
is_cellwise_constant() (ufl.NodalEnrichedElement method), 170
is_cellwise_constant() (ufl.RestrictedElement method), 170
is_cellwise_constant() (ufl.SpatialCoordinate method), 165
is_cellwise_constant() (ufl.variable.Label method), 158
is_globally_constant() (in module ufl.checks), 88
is_piecewise_linear_simplex_domain() (ufl.classes.Mesh method), 117
is_piecewise_linear_simplex_domain() (ufl.classes.MeshView method), 117
is_piecewise_linear_simplex_domain() (ufl.classes.TensorProductMesh method), 118
is_piecewise_linear_simplex_domain() (ufl.domain.Mesh method), 128
is_piecewise_linear_simplex_domain() (ufl.domain.MeshView method), 129
is_piecewise_linear_simplex_domain() (ufl.domain.TensorProductMesh method), 129
is_piecewise_linear_simplex_domain() (ufl.Mesh method), 164
is_piecewise_linear_simplex_domain() (ufl.MeshView method), 164
is_piecewise_linear_simplex_domain() (ufl.TensorProductMesh method), 165
is_post_handler() (in module ufl.algorithms.transformer), 72
is_python_scalar() (in module ufl.checks), 88
is_scalar_constant_expression() (in module ufl.checks), 88
is_simplex() (ufl.AbstractCell method), 163
is_simplex() (ufl.Cell method), 163
is_simplex() (ufl.cell.AbstractCell method), 87
is_simplex() (ufl.cell.Cell method), 87
is_simplex() (ufl.cell.TensorProductCell method), 87
is_simplex() (ufl.classes.AbstractCell method), 111
is_simplex() (ufl.classes.Cell method), 111
is_simplex() (ufl.classes.TensorProductCell method), 112
is_simplex() (ufl.TensorProductCell method), 164
is_true_ufl_scalar() (in module ufl.checks), 88
is_ufl_scalar() (in module ufl.checks), 88
iter_expressions() (in module ufl.algorithms.traversal), 73

J

Jacobian (class in ufl), 166
Jacobian (class in ufl.classes), 92
Jacobian (class in ufl.geometry), 140
jacobian() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
jacobian() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringAp method), 54
jacobian_determinant() (ufl.algorithms.apply_geometry_lowering.Geometry method), 54
jacobian_inverse() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
jacobian_inverse() (ufl.algorithms.apply_geometry_lowering.GeometryLow method), 54
JacobianDeterminant (class in ufl), 167
JacobianDeterminant (class in ufl.classes), 94
JacobianDeterminant (class in ufl.geometry), 140
JacobianInverse (class in ufl), 167
JacobianInverse (class in ufl.classes), 94
JacobianInverse (class in ufl.geometry), 140
join_domains() (in module ufl.domain), 129
jump() (in module ufl), 178
jump() (in module ufl.operators), 151

L

Label (class in ufl.classes), 101
Label (class in ufl.variable), 158
label() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
label() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
label() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
label() (ufl.classes.Variable method), 101
label() (ufl.variable.Variable method), 159
LE (class in ufl.classes), 108
LE (class in ufl.conditional), 123
le() (in module ufl), 176
le() (in module ufl.operators), 151
le() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
lhs() (in module ufl), 181
lhs() (in module ufl.formoperators), 133
linear_indexed_type() (ufl.algorithms.check_arities.ArityChecker method), 58
linear_indexed_type() (ufl.algorithms.formtransformations.PartExtracter method), 68
linear_operator() (ufl.algorithms.check_arities.ArityChecker method), 58
linear_operator() (ufl.algorithms.formtransformations.PartExtracter method), 68
list_tensor() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
list_tensor() (ufl.algorithms.check_arities.ArityChecker method), 58
list_tensor() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
list_tensor() (ufl.algorithms.expand_indices.IndexExpander method), 66
list_tensor() (ufl.algorithms.formtransformations.PartExtracter method), 68
ListTensor (class in ufl.classes), 99
ListTensor (class in ufl.tensors), 157
Ln (class in ufl.classes), 109
Ln (class in ufl.mathfunctions), 146
ln() (in module ufl), 176
ln() (in module ufl.operators), 151
ln() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
load_forms() (in module ufl.algorithms.formfiles), 66
load_ufl_file() (in module ufl.algorithms), 74
load_ufl_file() (in module ufl.algorithms.formfiles), 66
log() (ufl.log.Logger method), 145
Logger (class in ufl.log), 144
LowerCompoundAlgebra (class in ufl.algorithms.apply_algebra_lowering), 47
LT (class in ufl.classes), 108
LT (class in ufl.conditional), 124
lt() (in module ufl), 177
lt() (in module ufl.operators), 151
lt() (ufl.algorithms.comparison_checker.CheckComparisons method), 59

M

map_integrand_dags() (in module ufl.algorithms.map_integrands), 69
map_integrands() (in module ufl.algorithms.map_integrands), 69
mapping() (ufl.BrokenElement method), 171
mapping() (ufl.classes.BrokenElement method), 117
mapping() (ufl.classes.FiniteElement method), 113
mapping() (ufl.classes.FiniteElementBase method), 113
mapping() (ufl.classes.HCurlElement method), 116
mapping() (ufl.classes.HDivElement method), 116
mapping() (ufl.classes.MixedElement method), 114
mapping() (ufl.classes.RestrictedElement method), 115
mapping() (ufl.classes.TensorElement method), 114
mapping() (ufl.classes.TensorProductElement method), 116
mapping() (ufl.FiniteElement method), 168
mapping() (ufl.finiteelement.brokenelement.BrokenElement method), 77
mapping() (ufl.finiteelement.enrichedelement.EnrichedElementBase method), 78
mapping() (ufl.finiteelement.finiteelement.FiniteElement method), 79
mapping() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 79
mapping() (ufl.finiteelement.hdivcurl.HCurlElement method), 80
mapping() (ufl.finiteelement.hdivcurl.HDivElement method), 80
mapping() (ufl.finiteelement.mixedelement.MixedElement method), 81
mapping() (ufl.finiteelement.mixedelement.TensorElement method), 82
mapping() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
mapping() (ufl.finiteelement.tensorproductelement.TensorProductElement method), 83
mapping() (ufl.FiniteElementBase method), 167
mapping() (ufl.HCurlElement method), 171
mapping() (ufl.HDivElement method), 171
mapping() (ufl.MixedElement method), 169
mapping() (ufl.RestrictedElement method), 170
mapping() (ufl.TensorElement method), 169
mapping() (ufl.TensorProductElement method), 171
math_function() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
math_function() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64

MathFunction (class in ufl.classes), 109
MathFunction (class in ufl.mathfunctions), 146
Max() (in module ufl), 177
Max() (in module ufl.operators), 148
max_cell_edge_length() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
max_facet_edge_length() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
max_facet_edge_length() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
max_subdomain_ids() (ufl.classes.Form method), 121
max_subdomain_ids() (ufl.Form method), 179
max_subdomain_ids() (ufl.form.Form method), 131
max_value() (in module ufl), 177
max_value() (in module ufl.operators), 151
max_value() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 50
max_value() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
max_value() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
MaxCellEdgeLength (class in ufl), 165
MaxCellEdgeLength (class in ufl.classes), 96
MaxCellEdgeLength (class in ufl.geometry), 140
MaxFacetEdgeLength (class in ufl), 166
MaxFacetEdgeLength (class in ufl.classes), 97
MaxFacetEdgeLength (class in ufl.geometry), 140
MaxValue (class in ufl.classes), 109
MaxValue (class in ufl.conditional), 124
Measure (class in ufl), 180
Measure (class in ufl.classes), 119
Measure (class in ufl.measure), 147
measure_collisions() (in module ufl.exprequals), 130
measure_names() (in module ufl.measure), 148
MeasureProduct (class in ufl.classes), 119
MeasureProduct (class in ufl.measure), 147
MeasureSum (class in ufl.classes), 119
MeasureSum (class in ufl.measure), 148
merge_nonoverlapping_indices() (in module ufl.index_combination_utils), 142
merge_overlapping_indices() (in module ufl.index_combination_utils), 142
merge_unique_indices() (in module ufl.index_combination_utils), 143
Mesh (class in ufl), 164
Mesh (class in ufl.classes), 117
Mesh (class in ufl.domain), 128
MeshView (class in ufl), 164
MeshView (class in ufl.classes), 117
MeshView (class in ufl.domain), 129
metadata (ufl.algorithms.domain_analysis.IntegralData attribute), 61
metadata() (ufl.classes.Integral method), 120
metadata() (ufl.classes.Measure method), 119
metadata() (ufl.Integral method), 180
metadata() (ufl.integral.Integral method), 144
metadata() (ufl.Measure method), 180
metadata() (ufl.measure.Measure method), 147
metadata_equal() (in module ufl.protocols), 153
metadata_hash() (in module ufl.protocols), 153
Min() (in module ufl), 177
Min() (in module ufl.operators), 148
min_cell_edge_length() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
min_facet_edge_length() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
min_facet_edge_length() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
min_value() (in module ufl), 177
min_value() (in module ufl.operators), 151
min_value() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
min_value() (ufl.algorithms.comparison_checker.CheckComparisons method), 59
min_value() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
MinCellEdgeLength (class in ufl), 165
MinCellEdgeLength (class in ufl.classes), 96
MinCellEdgeLength (class in ufl.geometry), 141
MinFacetEdgeLength (class in ufl), 166
MinFacetEdgeLength (class in ufl.classes), 96
MinFacetEdgeLength (class in ufl.geometry), 141
MinValue (class in ufl.classes), 108
MinValue (class in ufl.conditional), 124
MixedElement (class in ufl), 168
MixedElement (class in ufl.classes), 113
MixedElement (class in ufl.finiteelement.mixedelement), 81
MixedFunctionSpace (class in ufl.classes), 118
MixedFunctionSpace (class in ufl.functionspace), 134
modulus() (ufl.classes.ComplexValue method), 99
modulus() (ufl.constantvalue.ComplexValue method), 125
multi_index() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
multi_index() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
multi_index() (ufl.algorithms.coordinate_derivative_helpers.CoordinateDerivativeRuleset method), 60
multi_index() (ufl.algorithms.estimate_degrees.SumDegreeEstimator 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), 74
MultiIndex (class in ufl.classes), 97

N

nabla_div() (in module ufl), 178
nabla_div() (in module ufl.operators), 151
nabla_div() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
nabla_div() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
nabla_grad() (in module ufl), 178
nabla_grad() (in module ufl.operators), 151
nabla_grad() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
nabla_grad() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
NablaDiv (class in ufl.classes), 107
NablaDiv (class in ufl.differentiation), 127
NablaGrad (class in ufl.classes), 107
NablaGrad (class in ufl.differentiation), 127
name (ufl.CellDiameter attribute), 165
name (ufl.CellNormal attribute), 166
name (ufl.CellVolume attribute), 165
name (ufl.Circumradius attribute), 165
name (ufl.classes.CellCoordinate attribute), 90
name (ufl.classes.CellDiameter attribute), 96
name (ufl.classes.CellEdgeVectors attribute), 93
name (ufl.classes.CellFacetJacobian attribute), 92
name (ufl.classes.CellFacetJacobianDeterminant attribute), 94
name (ufl.classes.CellFacetJacobianInverse attribute), 95
name (ufl.classes.CellFacetOrigin attribute), 91
name (ufl.classes.CellNormal attribute), 95
name (ufl.classes.CellOrientation attribute), 97
name (ufl.classes.CellOrigin attribute), 91
name (ufl.classes.CellVertices attribute), 93
name (ufl.classes.CellVolume attribute), 96
name (ufl.classes.Circumradius attribute), 96
name (ufl.classes.FacetArea attribute), 96
name (ufl.classes.FacetCoordinate attribute), 91
name (ufl.classes.FacetEdgeVectors attribute), 94
name (ufl.classes.FacetJacobian attribute), 92
name (ufl.classes.FacetJacobianDeterminant attribute), 94
name (ufl.classes.FacetJacobianInverse attribute), 95
name (ufl.classes.FacetNormal attribute), 95
name (ufl.classes.FacetOrientation attribute), 97
name (ufl.classes.FacetOrigin attribute), 91
name (ufl.classes.Jacobian attribute), 92
name (ufl.classes.JacobianDeterminant attribute), 94
name (ufl.classes.JacobianInverse attribute), 94
name (ufl.classes.MaxCellEdgeLength attribute), 96
name (ufl.classes.MaxFacetEdgeLength attribute), 97
name (ufl.classes.MinCellEdgeLength attribute), 96
name (ufl.classes.MinFacetEdgeLength attribute), 97
name (ufl.classes.QuadratureWeight attribute), 97
name (ufl.classes.ReferenceCellEdgeVectors attribute), 93
name (ufl.classes.ReferenceCellVolume attribute), 96
name (ufl.classes.ReferenceFacetEdgeVectors attribute), 93
name (ufl.classes.ReferenceFacetVolume attribute), 96
name (ufl.classes.ReferenceNormal attribute), 95
name (ufl.classes.SpatialCoordinate attribute), 90
name (ufl.FacetArea attribute), 166
name (ufl.FacetNormal attribute), 166
name (ufl.geometry.CellCoordinate attribute), 135
name (ufl.geometry.CellDiameter attribute), 135
name (ufl.geometry.CellEdgeVectors attribute), 135
name (ufl.geometry.CellFacetJacobian attribute), 135
name (ufl.geometry.CellFacetJacobianDeterminant attribute), 136
name (ufl.geometry.CellFacetJacobianInverse attribute), 136
name (ufl.geometry.CellFacetOrigin attribute), 136
name (ufl.geometry.CellNormal attribute), 136
name (ufl.geometry.CellOrientation attribute), 137
name (ufl.geometry.CellOrigin attribute), 137
name (ufl.geometry.CellVertices attribute), 137
name (ufl.geometry.CellVolume attribute), 137
name (ufl.geometry.Circumradius attribute), 137
name (ufl.geometry.FacetArea attribute), 137
name (ufl.geometry.FacetCoordinate attribute), 138
name (ufl.geometry.FacetEdgeVectors attribute), 138
name (ufl.geometry.FacetJacobian attribute), 138
name (ufl.geometry.FacetJacobianDeterminant attribute), 139
name (ufl.geometry.FacetJacobianInverse attribute), 139
name (ufl.geometry.FacetNormal attribute), 139
name (ufl.geometry.FacetOrientation attribute), 139
name (ufl.geometry.FacetOrigin attribute), 139
name (ufl.geometry.Jacobian attribute), 140
name (ufl.geometry.JacobianDeterminant attribute), 140
name (ufl.geometry.JacobianInverse attribute), 140
name (ufl.geometry.MaxCellEdgeLength attribute), 140
name (ufl.geometry.MaxFacetEdgeLength attribute), 141
name (ufl.geometry.MinCellEdgeLength attribute), 141
name (ufl.geometry.MinFacetEdgeLength attribute), 141
name (ufl.geometry.QuadratureWeight attribute), 141
name (ufl.geometry.ReferenceCellEdgeVectors attribute), 141
name (ufl.geometry.ReferenceCellVolume attribute), 141
name (ufl.geometry.ReferenceFacetEdgeVectors attribute), 141

name (ufl.geometry.ReferenceFacetVolume attribute), 142
name (ufl.geometry.ReferenceNormal attribute), 142
name (ufl.geometry.SpatialCoordinate attribute), 142
name (ufl.Jacobian attribute), 166
name (ufl.JacobianDeterminant attribute), 167
name (ufl.JacobianInverse attribute), 167
name (ufl.MaxCellEdgeLength attribute), 166
name (ufl.MaxFacetEdgeLength attribute), 166
name (ufl.MinCellEdgeLength attribute), 165
name (ufl.MinFacetEdgeLength attribute), 166
name (ufl.SpatialCoordinate attribute), 165
NE (class in ufl.classes), 108
NE (class in ufl.conditional), 124
ne() (in module ufl), 176
ne() (in module ufl.operators), 152
negative_restricted() (ufl.algorithms.balancing.BalanceModifiers method), 56
negative_restricted() (ufl.algorithms.check_arities.ArityChecker method), 58
negative_restricted() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
negative_restricted() (ufl.algorithms.formtransformations.PartExtracter method), 68
NegativeRestricted (class in ufl.classes), 105
NegativeRestricted (class in ufl.restriction), 154
NEWChangeToReferenceGrad (class in ufl.algorithms.change_to_reference), 57
NodalEnrichedElement (class in ufl), 170
NodalEnrichedElement (class in ufl.classes), 115
NodalEnrichedElement (class in ufl.finiteelement.enrichedelement), 78
non_differentiable_terminal() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
nonlinear_operator() (ufl.algorithms.check_arities.ArityChecker method), 58
nonrecursive_expr_equals() (in module ufl.exprequals), 130
Not() (in module ufl), 177
Not() (in module ufl.operators), 148
not_condition() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
NotCondition (class in ufl.classes), 108
NotCondition (class in ufl.conditional), 124
num_edges() (ufl.Cell method), 163
num_edges() (ufl.cell.Cell method), 87
num_edges() (ufl.cell.TensorProductCell method), 87
num_edges() (ufl.classes.Cell method), 111
num_edges() (ufl.classes.TensorProductCell method), 112
num_edges() (ufl.TensorProductCell method), 164
num_facet_edges() (ufl.Cell method), 163
num_facet_edges() (ufl.cell.Cell method), 87
num_facet_edges() (ufl.classes.Cell method), 111
num_facets() (ufl.Cell method), 163
num_facets() (ufl.cell.Cell method), 87
num_facets() (ufl.cell.TensorProductCell method), 87
num_facets() (ufl.classes.Cell method), 111
num_facets() (ufl.classes.TensorProductCell method), 112
num_facets() (ufl.TensorProductCell method), 164
num_restricted_sub_elements() (ufl.classes.RestrictedElement method), 115
num_restricted_sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
num_restricted_sub_elements() (ufl.RestrictedElement method), 170
num_sub_elements() (ufl.classes.FiniteElementBase method), 113
num_sub_elements() (ufl.classes.MixedElement method), 114
num_sub_elements() (ufl.classes.RestrictedElement method), 115
num_sub_elements() (ufl.classes.TensorProductElement method), 116
num_sub_elements() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
num_sub_elements() (ufl.finiteelement.mixedelement.MixedElement method), 81
num_sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
num_sub_elements() (ufl.finiteelement.tensorproductelement.TensorProduct method), 83
num_sub_elements() (ufl.FiniteElementBase method), 167
num_sub_elements() (ufl.MixedElement method), 169
num_sub_elements() (ufl.RestrictedElement method), 170
num_sub_elements() (ufl.TensorProductElement method), 171
num_vertices() (ufl.Cell method), 163
num_vertices() (ufl.cell.Cell method), 87
num_vertices() (ufl.cell.TensorProductCell method), 87
num_vertices() (ufl.classes.Cell method), 112
num_vertices() (ufl.classes.TensorProductCell method), 112
num_vertices() (ufl.TensorProductCell method), 164
number() (ufl.Argument method), 172
number() (ufl.argument.Argument method), 85
number() (ufl.classes.Argument method), 100
numpy2nestedlists() (in module ufl.tensors), 158

O

old_determinant_expr_3x3() (in module ufl.compound_expressions), 123

OLDChangeToReferenceGrad (class in ufl.algorithms.change_to_reference), 57
Operator (class in ufl.classes), 97
operator() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
operator() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 55
Or() (in module ufl), 177
Or() (in module ufl.operators), 149
OrCondition (class in ufl.classes), 108
OrCondition (class in ufl.conditional), 124
Outer (class in ufl.classes), 103
Outer (class in ufl.tensoralgebra), 156
outer() (in module ufl), 175
outer() (in module ufl.operators), 152
outer() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
outer() (ufl.algorithms.check_arities.ArityChecker method), 58
outer() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
outer() (ufl.algorithms.formtransformations.PartExtracter method), 68
override() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51

P

parstr() (in module ufl.precedence), 153
part() (ufl.Argument method), 172
part() (ufl.argument.Argument method), 85
part() (ufl.classes.Argument method), 100
PartExtracter (class in ufl.algorithms.formtransformations), 67
PermutationSymbol (class in ufl), 173
PermutationSymbol (class in ufl.classes), 99
PermutationSymbol (class in ufl.constantvalue), 125
perp() (in module ufl), 175
perp() (in module ufl.operators), 152
pop_level() (ufl.log.Logger method), 145
positive_restricted() (ufl.algorithms.balancing.BalanceModifier method), 56
positive_restricted() (ufl.algorithms.check_arities.ArityChecker method), 58
positive_restricted() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
positive_restricted() (ufl.algorithms.formtransformations.PartExtracter method), 68
PositiveRestricted (class in ufl.classes), 105
PositiveRestricted (class in ufl.restriction), 154
post_traversal() (in module ufl.algorithms), 75
Power (class in ufl.algebra), 84
Power (class in ufl.classes), 102
power() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
power() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
power() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
print_collisions() (in module ufl.exprequals), 130
print_visit_stack() (ufl.algorithms.Transformer method), 74
print_visit_stack() (ufl.algorithms.transformer.Transformer method), 72
Product (class in ufl.algebra), 84
Product (class in ufl.classes), 102
product() (in module ufl), 163
product() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
product() (ufl.algorithms.check_arities.ArityChecker method), 58
product() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
product() (ufl.algorithms.formtransformations.PartExtracter method), 68
pseudo_determinant_expr() (in module ufl.compound_expressions), 123
pseudo_inverse_expr() (in module ufl.compound_expressions), 123
purge_list_tensors() (in module ufl.algorithms), 73
purge_list_tensors() (in module ufl.algorithms.expand_indices), 66
push_level() (ufl.log.Logger method), 145

Q

quadrature_scheme() (ufl.classes.FiniteElementBase method), 113
quadrature_scheme() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
quadrature_scheme() (ufl.FiniteElementBase method), 167
quadrature_weight() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
QuadratureWeight (class in ufl.classes), 97
QuadratureWeight (class in ufl.geometry), 141

R

rank() (in module ufl), 174
rank() (in module ufl.operators), 152
read_lines_decoded() (in module ufl.algorithms.formfiles), 66
read_ufl_file() (in module ufl.algorithms.formfiles), 67
Real (class in ufl.algebra), 84
Real (class in ufl.classes), 103
real() (in module ufl), 175
real() (in module ufl.operators), 152
real() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51

real() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
real() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
real() (ufl.algorithms.formtransformations.PartExtracter method), 68
real() (ufl.algorithms.remove_complex_nodes.ComplexNodeReconstructor method), 70
real() (ufl.classes.ScalarValue method), 98
real() (ufl.constantvalue.ScalarValue method), 126
RealValue (class in ufl.classes), 99
RealValue (class in ufl.constantvalue), 125
rearrange_integrals_by_single_subdomains() (in module ufl.algorithms.domain_analysis), 62
reconstruct() (ufl.BrokenElement method), 171
reconstruct() (ufl.Cell method), 163
reconstruct() (ufl.cell.Cell method), 87
reconstruct() (ufl.cell.TensorProductCell method), 87
reconstruct() (ufl.classes.BrokenElement method), 117
reconstruct() (ufl.classes.Cell method), 112
reconstruct() (ufl.classes.FiniteElement method), 113
reconstruct() (ufl.classes.HCurlElement method), 116
reconstruct() (ufl.classes.HDivElement method), 116
reconstruct() (ufl.classes.Integral method), 120
reconstruct() (ufl.classes.Measure method), 119
reconstruct() (ufl.classes.MixedElement method), 114
reconstruct() (ufl.classes.RestrictedElement method), 115
reconstruct() (ufl.classes.TensorElement method), 114
reconstruct() (ufl.classes.TensorProductCell method), 112
reconstruct() (ufl.classes.TensorProductElement method), 116
reconstruct() (ufl.classes.VectorElement method), 114
reconstruct() (ufl.FiniteElement method), 168
reconstruct() (ufl.finiteelement.brokenelement.BrokenElement method), 77
reconstruct() (ufl.finiteelement.enrichedelement.EnrichedElementBase method), 78
reconstruct() (ufl.finiteelement.finiteelement.FiniteElement method), 79
reconstruct() (ufl.finiteelement.hdivcurl.HCurlElement method), 80
reconstruct() (ufl.finiteelement.hdivcurl.HDivElement method), 80
reconstruct() (ufl.finiteelement.mixedelement.MixedElement method), 81
reconstruct() (ufl.finiteelement.mixedelement.TensorElement method), 82
reconstruct() (ufl.finiteelement.mixedelement.VectorElement method), 82
reconstruct() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
reconstruct() (ufl.finiteelement.tensorproductelement.TensorProductElement method), 83
reconstruct() (ufl.HCurlElement method), 171
reconstruct() (ufl.HDivElement method), 171
reconstruct() (ufl.Integral method), 180
reconstruct() (ufl.integral.Integral method), 144
reconstruct() (ufl.Measure method), 180
reconstruct() (ufl.measure.Measure method), 147
reconstruct() (ufl.MixedElement method), 169
reconstruct() (ufl.RestrictedElement method), 170
reconstruct() (ufl.TensorElement method), 169
reconstruct() (ufl.TensorProductCell method), 164
reconstruct() (ufl.TensorProductElement method), 171
reconstruct() (ufl.VectorElement method), 169
reconstruct_form_from_integral_data() (in module ufl.algorithms.domain_analysis), 62
reconstruct_from_elements() (ufl.classes.MixedElement method), 114
reconstruct_from_elements() (ufl.finiteelement.mixedelement.MixedElement method), 81
reconstruct_from_elements() (ufl.MixedElement method), 169
reconstruct_variable() (ufl.algorithms.Transformer method), 74
reconstruct_variable() (ufl.algorithms.transformer.Transformer method), 72
recursive_expr_equals() (in module ufl.exprequals), 130
reference_cell_volume() (ufl.algorithms.apply_restrictions.RestrictionPropa method), 56
reference_curl() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 64
reference_div() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
reference_facet_volume() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
reference_grad() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRul method), 48
reference_grad() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRul method), 48
reference_grad() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatche method), 49
reference_grad() (ufl.algorithms.apply_derivatives.GateauxDerivativeRulese method), 49
reference_grad() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
reference_grad() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
reference_grad() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
reference_grad() (ufl.algorithms.balancing.BalanceModifiers method), 56
reference_grad() (ufl.algorithms.change_to_reference.NEWChangeToRefer method), 57
reference_grad() (ufl.algorithms.change_to_reference.OLDChangeToRefere method), 57

reference_grad() (ufl.algorithms.check_arities.ArityChecker method), 58
reference_grad() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
reference_value() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
reference_value() (ufl.algorithms.apply_derivatives.GateauxDerivativeRuleset method), 49
reference_value() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
reference_value() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
reference_value() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
reference_value() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
reference_value() (ufl.algorithms.balancing.BalanceModifier method), 56
reference_value() (ufl.algorithms.check_arities.ArityChecker method), 58
reference_value() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
reference_value_shape() (ufl.classes.FiniteElementBase method), 113
reference_value_shape() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
reference_value_shape() (ufl.FiniteElementBase method), 168
reference_value_size() (ufl.classes.FiniteElementBase method), 113
reference_value_size() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
reference_value_size() (ufl.FiniteElementBase method), 168
ReferenceCellEdgeVectors (class in ufl.classes), 92
ReferenceCellEdgeVectors (class in ufl.geometry), 141
ReferenceCellVolume (class in ufl.classes), 96
ReferenceCellVolume (class in ufl.geometry), 141
ReferenceCurl (class in ufl.classes), 107
ReferenceCurl (class in ufl.differentiation), 128
ReferenceDiv (class in ufl.classes), 107
ReferenceDiv (class in ufl.differentiation), 128
ReferenceFacetEdgeVectors (class in ufl.classes), 93
ReferenceFacetEdgeVectors (class in ufl.geometry), 141
ReferenceFacetVolume (class in ufl.classes), 96
ReferenceFacetVolume (class in ufl.geometry), 142
ReferenceGrad (class in ufl.classes), 106
ReferenceGrad (class in ufl.differentiation), 128
ReferenceGradRuleset (class in ufl.algorithms.apply_derivatives), 52
ReferenceNormal (class in ufl.classes), 95
ReferenceNormal (class in ufl.geometry), 142
ReferenceValue (class in ufl.classes), 111
ReferenceValue (class in ufl.referencevalue), 154
register_alias() (in module ufl.finiteelement.elementlist), 77
register_element() (in module ufl.finiteelement.elementlist), 172
register_element() (in module ufl.finiteelement.elementlist), 77
register_element2() (in module ufl.finiteelement.elementlist), 77
register_integral_type() (in module ufl), 181
register_integral_type() (in module ufl.measure), 148
relabel() (in module ufl), 174
relabel() (in module ufl.tensors), 158
remove_complex_nodes() (in module ufl.algorithms.remove_complex_nodes), 70
remove_indices() (in module ufl.index_combination_utils), 143
renumber_indices() (in module ufl.algorithms.renumbering), 70
replace() (in module ufl), 181
replace() (in module ufl.algorithms), 75
replace() (in module ufl.algorithms.replace), 71
replace_include_statements() (in module ufl.algorithms.formfiles), 67
replace_integral_domains() (in module ufl), 181
replace_integral_domains() (in module ufl.form), 132
Replacer (class in ufl.algorithms.replace), 70
reshape_to_nested_list() (in module ufl.algorithms.apply_function_pullbacks), 53
Restricted (class in ufl.classes), 105
Restricted (class in ufl.restriction), 154
restricted() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
restricted() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
restricted() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
restricted() (ufl.algorithms.change_to_reference.NEWChangeToReferenceC method), 57
restricted() (ufl.algorithms.check_restrictions.RestrictionChecker method), 59
restricted_sub_elements() (ufl.classes.RestrictedElement method), 115
restricted_sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
restricted_sub_elements() (ufl.RestrictedElement method), 170
RestrictedElement (class in ufl), 170
RestrictedElement (class in ufl.classes), 115
RestrictedElement (class in ufl.finiteelement.restrictedelement), 82
restriction_domain() (ufl.classes.RestrictedElement method), 115

restriction_domain() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
restriction_domain() (ufl.RestrictedElement method), 170
RestrictionChecker (class in ufl.algorithms.check_restrictions), 59
RestrictionPropagator (class in ufl.algorithms.apply_restrictions), 55
reuse() (ufl.algorithms.Transformer method), 74
reuse() (ufl.algorithms.transformer.Transformer method), 72
reuse_if_possible() (ufl.algorithms.Transformer method), 74
reuse_if_possible() (ufl.algorithms.transformer.Transformer method), 72
reuse_if_untouched() (ufl.algorithms.MultiFunction method), 75
reuse_if_untouched() (ufl.algorithms.Transformer method), 74
reuse_if_untouched() (ufl.algorithms.transformer.Transformer method), 72
reuse_variable() (ufl.algorithms.Transformer method), 74
reuse_variable() (ufl.algorithms.transformer.Transformer method), 72
ReuseTransformer (class in ufl.algorithms), 73
ReuseTransformer (class in ufl.algorithms.transformer), 71
rhs() (in module ufl), 181
rhs() (in module ufl.formoperators), 133
rot() (in module ufl), 178
rot() (in module ufl.operators), 152

S

scalar_value() (ufl.algorithms.expand_indices.IndexExpander method), 66
ScalarValue (class in ufl.classes), 98
ScalarValue (class in ufl.constantvalue), 125
sensitivity_rhs() (in module ufl), 182
sensitivity_rhs() (in module ufl.formoperators), 133
set_handler() (ufl.log.Logger method), 145
set_indent() (ufl.log.Logger method), 145
set_level() (ufl.log.Logger method), 145
set_list_item() (in module ufl.formoperators), 134
set_prefix() (ufl.log.Logger method), 145
shape() (in module ufl), 175
shape() (in module ufl.operators), 152
shortstr() (ufl.BrokenElement method), 172
shortstr() (ufl.classes.BrokenElement method), 117
shortstr() (ufl.classes.EnrichedElement method), 115
shortstr() (ufl.classes.FiniteElement method), 113
shortstr() (ufl.classes.HCurlElement method), 116
shortstr() (ufl.classes.HDivElement method), 116
shortstr() (ufl.classes.MixedElement method), 114
shortstr() (ufl.classes.NodalEnrichedElement method), 115
shortstr() (ufl.classes.RestrictedElement method), 115
shortstr() (ufl.classes.TensorElement method), 114
shortstr() (ufl.classes.TensorProductElement method), 116
shortstr() (ufl.classes.VectorElement method), 114
shortstr() (ufl.EnrichedElement method), 170
shortstr() (ufl.FiniteElement method), 168
shortstr() (ufl.finiteelement.brokenelement.BrokenElement method), 77
shortstr() (ufl.finiteelement.enrichedelement.EnrichedElement method), 78
shortstr() (ufl.finiteelement.enrichedelement.NodalEnrichedElement method), 78
shortstr() (ufl.finiteelement.finiteelement.FiniteElement method), 79
shortstr() (ufl.finiteelement.hdivcurl.HCurlElement method), 80
shortstr() (ufl.finiteelement.hdivcurl.HDivElement method), 80
shortstr() (ufl.finiteelement.mixedelement.MixedElement method), 81
shortstr() (ufl.finiteelement.mixedelement.TensorElement method), 82
shortstr() (ufl.finiteelement.mixedelement.VectorElement method), 82
shortstr() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
shortstr() (ufl.finiteelement.tensorproductelement.TensorProductElement method), 83
shortstr() (ufl.HCurlElement method), 171
shortstr() (ufl.HDivElement method), 171
shortstr() (ufl.MixedElement method), 169
shortstr() (ufl.NodalEnrichedElement method), 170
shortstr() (ufl.RestrictedElement method), 170
shortstr() (ufl.TensorElement method), 169
shortstr() (ufl.TensorProductElement method), 171
shortstr() (ufl.VectorElement method), 169
show_elements() (in module ufl), 172
show_elements() (in module ufl.finiteelement.elementlist), 77
side() (ufl.classes.Restricted method), 105
side() (ufl.restriction.Restricted method), 154
sign() (in module ufl), 177
sign() (in module ufl.operators), 152
sign() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
signature() (ufl.classes.Form method), 121
signature() (ufl.Form method), 179
signature() (ufl.form.Form method), 131
simplex() (in module ufl.cell), 88
Sin (class in ufl.classes), 109
Sin (class in ufl.mathfunctions), 147
sin() (in module ufl), 176
sin() (in module ufl.operators), 152

sin() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
Sinh (class in ufl.classes), 109
Sinh (class in ufl.mathfunctions), 147
sinh() (in module ufl), 176
sinh() (in module ufl.operators), 152
sinh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
Skew (class in ufl.classes), 104
Skew (class in ufl.tensoralgebra), 156
skew() (in module ufl), 175
skew() (in module ufl.operators), 152
skew() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
skew() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
sobolev_space() (ufl.classes.FiniteElement method), 113
sobolev_space() (ufl.classes.HCurlElement method), 116
sobolev_space() (ufl.classes.HDivElement method), 116
sobolev_space() (ufl.classes.TensorProductElement method), 116
sobolev_space() (ufl.FiniteElement method), 168
sobolev_space() (ufl.finiteelement.enrichedelement.EnrichedElement method), 78
sobolev_space() (ufl.finiteelement.finiteelement.FiniteElement method), 79
sobolev_space() (ufl.finiteelement.hdivcurl.HCurlElement method), 80
sobolev_space() (ufl.finiteelement.hdivcurl.HDivElement method), 80
sobolev_space() (ufl.finiteelement.tensorproductelement.TensorProductElement method), 83
sobolev_space() (ufl.HCurlElement method), 171
sobolev_space() (ufl.HDivElement method), 171
sobolev_space() (ufl.TensorProductElement method), 171
SobolevSpace (class in ufl.sobolevspace), 154
sort_domains() (in module ufl.domain), 129
sort_elements() (in module ufl.algorithms), 73
sort_elements() (in module ufl.algorithms.analysis), 47
sorted_expr() (in module ufl.sorting), 155
sorted_expr_sum() (in module ufl.sorting), 155
spatial_coordinate() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleset method), 48
spatial_coordinate() (ufl.algorithms.apply_derivatives.GradRuleset method), 51
spatial_coordinate() (ufl.algorithms.apply_derivatives.ReferenceGradRuleset method), 52
spatial_coordinate() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringApplier method), 54
spatial_coordinate() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
spatial_coordinate() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
SpatialCoordinate (class in ufl), 165
SpatialCoordinate (class in ufl.classes), 90
SpatialCoordinate (class in ufl.geometry), 142
split() (in module ufl), 173
split() (in module ufl.split_functions), 155
split() (ufl.algorithms.FormSplitter method), 76
split() (ufl.algorithms.formsplitter.FormSplitter method), 67
Sqrt (class in ufl.classes), 109
Sqrt (class in ufl.mathfunctions), 147
sqrt() (in module ufl), 175
sqrt() (in module ufl.operators), 152
sqrt() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
sqrt() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
strip_coordinate_derivatives() (in module ufl.algorithms.coordinate_derivative_helpers), 61
strip_variables() (in module ufl.algorithms), 75
strip_variables() (in module ufl.algorithms.transformer), 72
sub_cells() (ufl.cell.TensorProductCell method), 87
sub_cells() (ufl.classes.TensorProductCell method), 112
sub_cells() (ufl.TensorProductCell method), 164
sub_element() (ufl.classes.RestrictedElement method), 116
sub_element() (ufl.finiteelement.restrictedelement.RestrictedElement method), 82
sub_element() (ufl.RestrictedElement method), 170
sub_elements() (ufl.classes.FiniteElementBase method), 113
sub_elements() (ufl.classes.MixedElement method), 114
sub_elements() (ufl.classes.RestrictedElement method), 116
sub_elements() (ufl.classes.TensorProductElement method), 116
sub_elements() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
sub_elements() (ufl.finiteelement.mixedelement.MixedElement method), 81
sub_elements() (ufl.finiteelement.restrictedelement.RestrictedElement method), 83
sub_elements() (ufl.finiteelement.tensorproductelement.TensorProductElement method), 83
sub_elements() (ufl.FiniteElementBase method), 168
sub_elements() (ufl.MixedElement method), 169
sub_elements() (ufl.RestrictedElement method), 170
sub_elements() (ufl.TensorProductElement method), 171
sub_elements_with_mappings() (in module ufl.algorithms.apply_function_pullbacks), 53
sub_entities() (ufl.classes.MeasureProduct method), 120
sub_measures() (ufl.measure.MeasureProduct method),

148
subdomain_data() (ufl.classes.Form method), 121
subdomain_data() (ufl.classes.Integral method), 120
subdomain_data() (ufl.classes.Measure method), 119
subdomain_data() (ufl.Form method), 179
subdomain_data() (ufl.form.Form method), 131
subdomain_data() (ufl.Integral method), 180
subdomain_data() (ufl.integral.Integral method), 144
subdomain_data() (ufl.Measure method), 180
subdomain_data() (ufl.measure.Measure method), 147
subdomain_id (ufl.algorithms.domain_analysis.IntegralData attribute), 61
subdomain_id() (ufl.classes.Integral method), 120
subdomain_id() (ufl.classes.Measure method), 119
subdomain_id() (ufl.Integral method), 180
subdomain_id() (ufl.integral.Integral method), 144
subdomain_id() (ufl.Measure method), 180
subdomain_id() (ufl.measure.Measure method), 147
Sum (class in ufl.algebra), 85
Sum (class in ufl.classes), 102
sum() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
sum() (ufl.algorithms.check_arities.ArityChecker method), 58
sum() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
sum() (ufl.algorithms.formtransformations.PartExtracter method), 68
SumDegreeEstimator (class in ufl.algorithms.estimate_degrees), 62
Sym (class in ufl.classes), 104
Sym (class in ufl.tensoralgebra), 156
sym() (in module ufl), 175
sym() (in module ufl.operators), 152
sym() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
sym() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
symmetry() (ufl.classes.FiniteElementBase method), 113
symmetry() (ufl.classes.MixedElement method), 114
symmetry() (ufl.classes.RestrictedElement method), 116
symmetry() (ufl.classes.TensorElement method), 114
symmetry() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
symmetry() (ufl.finiteelement.mixedelement.MixedElement method), 81
symmetry() (ufl.finiteelement.mixedelement.TensorElement method), 82
symmetry() (ufl.finiteelement.restrictedelement.RestrictedElement method), 83
symmetry() (ufl.FiniteElementBase method), 168
symmetry() (ufl.MixedElement method), 169
symmetry() (ufl.RestrictedElement method), 170
symmetry() (ufl.TensorElement method), 169
system() (in module ufl), 182
system() (in module ufl.formoperators), 134

T

T (ufl.classes.Expr attribute), 89
Tan (class in ufl.classes), 109
Tan (class in ufl.mathfunctions), 147
tan() (in module ufl), 176
tan() (in module ufl.operators), 152
tan() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
Tanh (class in ufl.classes), 110
Tanh (class in ufl.mathfunctions), 147
tanh() (in module ufl), 176
tanh() (in module ufl.operators), 152
tanh() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
tear() (in module ufl.algorithms.elementtransformations), 62
TensorConstant() (in module ufl), 173
TensorConstant() (in module ufl.coefficient), 122
TensorElement (class in ufl), 169
TensorElement (class in ufl.classes), 114
TensorElement (class in ufl.finiteelement.mixedelement), 81
TensorProductCell (class in ufl), 163
TensorProductCell (class in ufl.cell), 87
TensorProductCell (class in ufl.classes), 112
TensorProductElement (class in ufl), 170
TensorProductElement (class in ufl.classes), 116
TensorProductElement (class in ufl.finiteelement.tensorproductelement), 83
TensorProductFunctionSpace (class in ufl.classes), 118
TensorProductFunctionSpace (class in ufl.functionspace), 134
TensorProductMesh (class in ufl), 165
TensorProductMesh (class in ufl.classes), 117
TensorProductMesh (class in ufl.domain), 129
Terminal (class in ufl.classes), 89
terminal() (ufl.algorithms.apply_derivatives.CoordinateDerivativeRuleDispa method), 48
terminal() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
terminal() (ufl.algorithms.apply_function_pullbacks.FunctionPullbackAppli method), 53
terminal() (ufl.algorithms.apply_geometry_lowering.GeometryLoweringAp method), 54
terminal() (ufl.algorithms.apply_restrictions.DefaultRestrictionApplier method), 55
terminal() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
terminal() (ufl.algorithms.balancing.BalanceModifiers method), 56

terminal() (ufl.algorithms.change_to_reference.NEWChangeToReferenceGrad method), 57
terminal() (ufl.algorithms.change_to_reference.OLDChangeToReferenceGrad method), 57
terminal() (ufl.algorithms.check_arities.ArityChecker method), 58
terminal() (ufl.algorithms.comparison_checker.CheckComparisons method), 60
terminal() (ufl.algorithms.coordinate_derivative_helpers.CoordinateDerivativeRuleDispatcher method), 60
terminal() (ufl.algorithms.expand_indices.IndexExpander method), 66
terminal() (ufl.algorithms.formtransformations.PartExtracter method), 68
terminal() (ufl.algorithms.remove_complex_nodes.ComplexNodeRemoval method), 70
terminal() (ufl.algorithms.replace.Replacer method), 71
terminal() (ufl.algorithms.ReuseTransformer method), 74
terminal() (ufl.algorithms.Transformer method), 74
terminal() (ufl.algorithms.transformer.CopyTransformer method), 71
terminal() (ufl.algorithms.transformer.ReuseTransformer method), 71
terminal() (ufl.algorithms.transformer.Transformer method), 72
TestFunction() (in module ufl), 172
TestFunction() (in module ufl.argument), 85
TestFunction() (in module ufl.classes), 119
TestFunctions() (in module ufl), 173
TestFunctions() (in module ufl.argument), 85
TestFunctions() (in module ufl.classes), 119
topological_dimension() (ufl.AbstractCell method), 163
topological_dimension() (ufl.AbstractDomain method), 164
topological_dimension() (ufl.cell.AbstractCell method), 87
topological_dimension() (ufl.classes.AbstractCell method), 111
topological_dimension() (ufl.classes.AbstractDomain method), 117
topological_dimension() (ufl.domain.AbstractDomain method), 128
tr() (in module ufl), 175
tr() (in module ufl.operators), 152
Trace (class in ufl.classes), 104
Trace (class in ufl.tensoralgebra), 156
trace() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
trace() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
Transformer (class in ufl.algorithms), 74
Transformer (class in ufl.algorithms.transformer), 72
transpose() (in module ufl), 175
transpose() (in module ufl.operators), 152
Transposed (class in ufl.classes), 103
Transposed (class in ufl.tensoralgebra), 156
transposed() (ufl.algorithms.apply_algebra_lowering.LowerCompoundAlgebra method), 47
transposed() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
tree_format() (in module ufl.algorithms), 77
TrialFunction() (in module ufl), 172
TrialFunction() (in module ufl.argument), 86
TrialFunction() (in module ufl.classes), 119
TrialFunctions() (in module ufl), 173
TrialFunctions() (in module ufl.argument), 86
TrialFunctions() (in module ufl.classes), 119

U

ufl (module), 159
ufl.algebra (module), 83
ufl.algorithms (module), 73
ufl.algorithms.ad (module), 46
ufl.algorithms.analysis (module), 46
ufl.algorithms.apply_algebra_lowering (module), 47
ufl.algorithms.apply_derivatives (module), 48
ufl.algorithms.apply_function_pullbacks (module), 53
ufl.algorithms.apply_geometry_lowering (module), 53
ufl.algorithms.apply_integral_scaling (module), 54
ufl.algorithms.apply_restrictions (module), 54
ufl.algorithms.balancing (module), 56
ufl.algorithms.change_to_reference (module), 57
ufl.algorithms.check_arities (module), 58
ufl.algorithms.check_restrictions (module), 59
ufl.algorithms.checks (module), 59
ufl.algorithms.comparison_checker (module), 59
ufl.algorithms.compute_form_data (module), 60
ufl.algorithms.coordinate_derivative_helpers (module), 60
ufl.algorithms.domain_analysis (module), 61
ufl.algorithms.elementtransformations (module), 62
ufl.algorithms.estimate_degrees (module), 62
ufl.algorithms.expand_compounds (module), 65
ufl.algorithms.expand_indices (module), 65
ufl.algorithms.formdata (module), 66
ufl.algorithms.formfiles (module), 66
ufl.algorithms.formsplitter (module), 67
ufl.algorithms.formtransformations (module), 67
ufl.algorithms.map_integrands (module), 69
ufl.algorithms.multifunction (module), 70
ufl.algorithms.remove_complex_nodes (module), 70
ufl.algorithms.renumbering (module), 70
ufl.algorithms.replace (module), 70
ufl.algorithms.signature (module), 71
ufl.algorithms.transformer (module), 71
ufl.algorithms.traversal (module), 73
ufl.argument (module), 85
ufl.assertions (module), 86

ufl.averaging (module), 86
ufl.cell (module), 86
ufl.checks (module), 88
ufl.classes (module), 88
ufl.coefficient (module), 121
ufl.compound_expressions (module), 122
ufl.conditional (module), 123
ufl.constantvalue (module), 124
ufl.differentiation (module), 126
ufl.domain (module), 128
ufl.equation (module), 129
ufl.exprcontainers (module), 130
ufl.exprequals (module), 130
ufl.exproperators (module), 131
ufl.finiteelement (module), 83
ufl.finiteelement.brokenelement (module), 77
ufl.finiteelement.elementlist (module), 77
ufl.finiteelement.enrichedelement (module), 77
ufl.finiteelement.facetelement (module), 78
ufl.finiteelement.finiteelement (module), 78
ufl.finiteelement.finiteelementbase (module), 79
ufl.finiteelement.hdivcurl (module), 80
ufl.finiteelement.interiorelement (module), 81
ufl.finiteelement.mixedelement (module), 81
ufl.finiteelement.restrictedelement (module), 82
ufl.finiteelement.tensorproductelement (module), 83
ufl.form (module), 131
ufl.formoperators (module), 132
ufl.functionspace (module), 134
ufl.geometry (module), 135
ufl.index_combination_utils (module), 142
ufl.indexed (module), 143
ufl.indexsum (module), 143
ufl.integral (module), 143
ufl.log (module), 144
ufl.mathfunctions (module), 145
ufl.measure (module), 147
ufl.objects (module), 148
ufl.operators (module), 148
ufl.permutation (module), 153
ufl.precedence (module), 153
ufl.protocols (module), 153
ufl.referencevalue (module), 154
ufl.restriction (module), 154
ufl.sobolevspace (module), 154
ufl.sorting (module), 155
ufl.split_functions (module), 155
ufl.tensoralgebra (module), 155
ufl.tensors (module), 157
ufl.variable (module), 158
ufl2ufl() (in module ufl.algorithms.transformer), 72
ufl2uflcopy() (in module ufl.algorithms.transformer), 73
ufl_assert() (in module ufl.assertions), 86
ufl_cargo() (ufl.classes.Mesh method), 117
ufl_cargo() (ufl.domain.Mesh method), 128
ufl_cargo() (ufl.Mesh method), 164
ufl_cell() (ufl.classes.Form method), 121
ufl_cell() (ufl.classes.Mesh method), 117
ufl_cell() (ufl.classes.MeshView method), 117
ufl_cell() (ufl.classes.TensorProductMesh method), 118
ufl_cell() (ufl.domain.Mesh method), 128
ufl_cell() (ufl.domain.MeshView method), 129
ufl_cell() (ufl.domain.TensorProductMesh method), 129
ufl_cell() (ufl.Form method), 179
ufl_cell() (ufl.form.Form method), 131
ufl_cell() (ufl.Mesh method), 164
ufl_cell() (ufl.MeshView method), 164
ufl_cell() (ufl.TensorProductMesh method), 165
ufl_coordinate_element() (ufl.classes.Mesh method), 117
ufl_coordinate_element() (ufl.classes.TensorProductMesh method), 118
ufl_coordinate_element() (ufl.domain.Mesh method), 128
ufl_coordinate_element() (ufl.domain.TensorProductMesh method), 129
ufl_coordinate_element() (ufl.Mesh method), 164
ufl_coordinate_element() (ufl.TensorProductMesh method), 165
ufl_disable_profiling() (ufl.classes.Expr static method), 89
ufl_domain() (ufl.Argument method), 172
ufl_domain() (ufl.argument.Argument method), 85
ufl_domain() (ufl.classes.Argument method), 100
ufl_domain() (ufl.classes.Coefficient method), 100
ufl_domain() (ufl.classes.Expr method), 89
ufl_domain() (ufl.classes.Form method), 121
ufl_domain() (ufl.classes.FunctionSpace method), 118
ufl_domain() (ufl.classes.Integral method), 120
ufl_domain() (ufl.classes.Measure method), 119
ufl_domain() (ufl.classes.MixedFunctionSpace method), 118
ufl_domain() (ufl.Coefficient method), 173
ufl_domain() (ufl.coefficient.Coefficient method), 121
ufl_domain() (ufl.Form method), 179
ufl_domain() (ufl.form.Form method), 132
ufl_domain() (ufl.FunctionSpace method), 172
ufl_domain() (ufl.functionspace.FunctionSpace method), 134
ufl_domain() (ufl.functionspace.MixedFunctionSpace method), 134
ufl_domain() (ufl.Integral method), 180
ufl_domain() (ufl.integral.Integral method), 144
ufl_domain() (ufl.Measure method), 180
ufl_domain() (ufl.measure.Measure method), 147
ufl_domains() (ufl.Argument method), 172
ufl_domains() (ufl.argument.Argument method), 85
ufl_domains() (ufl.classes.Argument method), 100

ufl_domains() (ufl.classes.Coefficient method), 100
ufl_domains() (ufl.classes.ConstantValue method), 98
ufl_domains() (ufl.classes.Expr method), 89
ufl_domains() (ufl.classes.ExprMapping method), 105
ufl_domains() (ufl.classes.Form method), 121
ufl_domains() (ufl.classes.FunctionSpace method), 118
ufl_domains() (ufl.classes.GeometricQuantity method), 90
ufl_domains() (ufl.classes.Label method), 101
ufl_domains() (ufl.classes.MixedFunctionSpace method), 118
ufl_domains() (ufl.classes.MultiIndex method), 97
ufl_domains() (ufl.classes.Terminal method), 89
ufl_domains() (ufl.classes.Variable method), 101
ufl_domains() (ufl.Coefficient method), 173
ufl_domains() (ufl.coefficient.Coefficient method), 121
ufl_domains() (ufl.constantvalue.ConstantValue method), 125
ufl_domains() (ufl.exprcontainers.ExprMapping method), 130
ufl_domains() (ufl.Form method), 179
ufl_domains() (ufl.form.Form method), 132
ufl_domains() (ufl.FunctionSpace method), 172
ufl_domains() (ufl.functionspace.FunctionSpace method), 134
ufl_domains() (ufl.functionspace.MixedFunctionSpace method), 134
ufl_domains() (ufl.geometry.GeometricQuantity method), 140
ufl_domains() (ufl.variable.Label method), 158
ufl_domains() (ufl.variable.Variable method), 159
ufl_element() (ufl.Argument method), 172
ufl_element() (ufl.argument.Argument method), 85
ufl_element() (ufl.classes.Argument method), 100
ufl_element() (ufl.classes.Coefficient method), 101
ufl_element() (ufl.classes.FunctionSpace method), 118
ufl_element() (ufl.classes.MixedFunctionSpace method), 118
ufl_element() (ufl.Coefficient method), 173
ufl_element() (ufl.coefficient.Coefficient method), 122
ufl_element() (ufl.FunctionSpace method), 172
ufl_element() (ufl.functionspace.FunctionSpace method), 134
ufl_element() (ufl.functionspace.MixedFunctionSpace method), 134
ufl_enable_profiling() (ufl.classes.Expr static method), 89
ufl_free_indices (ufl.algebra.Abs attribute), 83
ufl_free_indices (ufl.algebra.Conj attribute), 84
ufl_free_indices (ufl.algebra.Division attribute), 84
ufl_free_indices (ufl.algebra.Imag attribute), 84
ufl_free_indices (ufl.algebra.Power attribute), 84
ufl_free_indices (ufl.algebra.Product attribute), 84
ufl_free_indices (ufl.algebra.Real attribute), 85
ufl_free_indices (ufl.algebra.Sum attribute), 85
ufl_free_indices (ufl.averaging.CellAvg attribute), 86
ufl_free_indices (ufl.averaging.FacetAvg attribute), 86
ufl_free_indices (ufl.classes.Abs attribute), 102
ufl_free_indices (ufl.classes.Atan2 attribute), 110
ufl_free_indices (ufl.classes.BesselFunction attribute), 110
ufl_free_indices (ufl.classes.CellAvg attribute), 110
ufl_free_indices (ufl.classes.CoefficientDerivative attribute), 106
ufl_free_indices (ufl.classes.Cofactor attribute), 104
ufl_free_indices (ufl.classes.ComponentTensor attribute), 100
ufl_free_indices (ufl.classes.Condition attribute), 107
ufl_free_indices (ufl.classes.Conditional attribute), 108
ufl_free_indices (ufl.classes.Conj attribute), 103
ufl_free_indices (ufl.classes.CoordinateDerivative attribute), 106
ufl_free_indices (ufl.classes.Cross attribute), 104
ufl_free_indices (ufl.classes.Curl attribute), 107
ufl_free_indices (ufl.classes.Determinant attribute), 104
ufl_free_indices (ufl.classes.Deviatoric attribute), 104
ufl_free_indices (ufl.classes.Div attribute), 107
ufl_free_indices (ufl.classes.Division attribute), 102
ufl_free_indices (ufl.classes.Dot attribute), 103
ufl_free_indices (ufl.classes.ExprList attribute), 105
ufl_free_indices (ufl.classes.ExprMapping attribute), 105
ufl_free_indices (ufl.classes.FacetAvg attribute), 111
ufl_free_indices (ufl.classes.Grad attribute), 106
ufl_free_indices (ufl.classes.Imag attribute), 103
ufl_free_indices (ufl.classes.Indexed attribute), 99
ufl_free_indices (ufl.classes.IndexSum attribute), 105
ufl_free_indices (ufl.classes.Inner attribute), 103
ufl_free_indices (ufl.classes.Inverse attribute), 104
ufl_free_indices (ufl.classes.Label attribute), 101
ufl_free_indices (ufl.classes.ListTensor attribute), 100
ufl_free_indices (ufl.classes.MathFunction attribute), 109
ufl_free_indices (ufl.classes.MaxValue attribute), 109
ufl_free_indices (ufl.classes.MinValue attribute), 109
ufl_free_indices (ufl.classes.MultiIndex attribute), 98
ufl_free_indices (ufl.classes.NablaDiv attribute), 107
ufl_free_indices (ufl.classes.NablaGrad attribute), 107
ufl_free_indices (ufl.classes.Outer attribute), 103
ufl_free_indices (ufl.classes.Power attribute), 102
ufl_free_indices (ufl.classes.Product attribute), 102
ufl_free_indices (ufl.classes.Real attribute), 103
ufl_free_indices (ufl.classes.RealValue attribute), 99
ufl_free_indices (ufl.classes.ReferenceCurl attribute), 107
ufl_free_indices (ufl.classes.ReferenceDiv attribute), 107
ufl_free_indices (ufl.classes.ReferenceGrad attribute), 106
ufl_free_indices (ufl.classes.ReferenceValue attribute), 111
ufl_free_indices (ufl.classes.Restricted attribute), 105
ufl_free_indices (ufl.classes.ScalarValue attribute), 98

ufl_free_indices (ufl.classes.Skew attribute), 104
ufl_free_indices (ufl.classes.Sum attribute), 102
ufl_free_indices (ufl.classes.Sym attribute), 104
ufl_free_indices (ufl.classes.Terminal attribute), 89
ufl_free_indices (ufl.classes.Trace attribute), 104
ufl_free_indices (ufl.classes.Transposed attribute), 103
ufl_free_indices (ufl.classes.Variable attribute), 101
ufl_free_indices (ufl.classes.VariableDerivative attribute), 106
ufl_free_indices (ufl.classes.Zero attribute), 98
ufl_free_indices (ufl.conditional.Condition attribute), 123
ufl_free_indices (ufl.conditional.Conditional attribute), 123
ufl_free_indices (ufl.conditional.MaxValue attribute), 124
ufl_free_indices (ufl.conditional.MinValue attribute), 124
ufl_free_indices (ufl.constantvalue.RealValue attribute), 125
ufl_free_indices (ufl.constantvalue.ScalarValue attribute), 126
ufl_free_indices (ufl.constantvalue.Zero attribute), 126
ufl_free_indices (ufl.differentiation.CoefficientDerivative attribute), 126
ufl_free_indices (ufl.differentiation.CoordinateDerivative attribute), 127
ufl_free_indices (ufl.differentiation.Curl attribute), 127
ufl_free_indices (ufl.differentiation.Div attribute), 127
ufl_free_indices (ufl.differentiation.Grad attribute), 127
ufl_free_indices (ufl.differentiation.NablaDiv attribute), 127
ufl_free_indices (ufl.differentiation.NablaGrad attribute), 127
ufl_free_indices (ufl.differentiation.ReferenceCurl attribute), 128
ufl_free_indices (ufl.differentiation.ReferenceDiv attribute), 128
ufl_free_indices (ufl.differentiation.ReferenceGrad attribute), 128
ufl_free_indices (ufl.differentiation.VariableDerivative attribute), 128
ufl_free_indices (ufl.exprcontainers.ExprList attribute), 130
ufl_free_indices (ufl.exprcontainers.ExprMapping attribute), 130
ufl_free_indices (ufl.indexed.Indexed attribute), 143
ufl_free_indices (ufl.indexsum.IndexSum attribute), 143
ufl_free_indices (ufl.mathfunctions.Atan2 attribute), 146
ufl_free_indices (ufl.mathfunctions.BesselFunction attribute), 146
ufl_free_indices (ufl.mathfunctions.MathFunction attribute), 146
ufl_free_indices (ufl.referencevalue.ReferenceValue attribute), 154
ufl_free_indices (ufl.restriction.Restricted attribute), 154
ufl_free_indices (ufl.tensoralgebra.Cofactor attribute), 155
ufl_free_indices (ufl.tensoralgebra.Cross attribute), 155
ufl_free_indices (ufl.tensoralgebra.Determinant attribute), 155
ufl_free_indices (ufl.tensoralgebra.Deviatoric attribute), 155
ufl_free_indices (ufl.tensoralgebra.Dot attribute), 156
ufl_free_indices (ufl.tensoralgebra.Inner attribute), 156
ufl_free_indices (ufl.tensoralgebra.Inverse attribute), 156
ufl_free_indices (ufl.tensoralgebra.Outer attribute), 156
ufl_free_indices (ufl.tensoralgebra.Skew attribute), 156
ufl_free_indices (ufl.tensoralgebra.Sym attribute), 156
ufl_free_indices (ufl.tensoralgebra.Trace attribute), 156
ufl_free_indices (ufl.tensoralgebra.Transposed attribute), 156
ufl_free_indices (ufl.tensors.ComponentTensor attribute), 157
ufl_free_indices (ufl.tensors.ListTensor attribute), 157
ufl_free_indices (ufl.variable.Label attribute), 158
ufl_free_indices (ufl.variable.Variable attribute), 159
ufl_function_space() (ufl.Argument method), 172
ufl_function_space() (ufl.argument.Argument method), 85
ufl_function_space() (ufl.classes.Argument method), 100
ufl_function_space() (ufl.classes.Coefficient method), 101
ufl_function_space() (ufl.Coefficient method), 173
ufl_function_space() (ufl.coefficient.Coefficient method), 122
ufl_id() (ufl.classes.Mesh method), 117
ufl_id() (ufl.classes.MeshView method), 117
ufl_id() (ufl.classes.TensorProductMesh method), 118
ufl_id() (ufl.domain.Mesh method), 129
ufl_id() (ufl.domain.MeshView method), 129
ufl_id() (ufl.domain.TensorProductMesh method), 129
ufl_id() (ufl.Mesh method), 164
ufl_id() (ufl.MeshView method), 164
ufl_id() (ufl.TensorProductMesh method), 165
ufl_index_dimensions (ufl.algebra.Abs attribute), 83
ufl_index_dimensions (ufl.algebra.Conj attribute), 84
ufl_index_dimensions (ufl.algebra.Division attribute), 84
ufl_index_dimensions (ufl.algebra.Imag attribute), 84
ufl_index_dimensions (ufl.algebra.Power attribute), 84
ufl_index_dimensions (ufl.algebra.Product attribute), 84
ufl_index_dimensions (ufl.algebra.Real attribute), 85
ufl_index_dimensions (ufl.algebra.Sum attribute), 85
ufl_index_dimensions (ufl.averaging.CellAvg attribute), 86
ufl_index_dimensions (ufl.averaging.FacetAvg attribute), 86
ufl_index_dimensions (ufl.classes.Abs attribute), 102
ufl_index_dimensions (ufl.classes.Atan2 attribute), 110
ufl_index_dimensions (ufl.classes.BesselFunction attribute), 110

ufl_index_dimensions (ufl.classes.CellAvg attribute), 110
ufl_index_dimensions (ufl.classes.CoefficientDerivative attribute), 106
ufl_index_dimensions (ufl.classes.Cofactor attribute), 104
ufl_index_dimensions (ufl.classes.ComponentTensor attribute), 100
ufl_index_dimensions (ufl.classes.Condition attribute), 107
ufl_index_dimensions (ufl.classes.Conditional attribute), 108
ufl_index_dimensions (ufl.classes.Conj attribute), 103
ufl_index_dimensions (ufl.classes.CoordinateDerivative attribute), 106
ufl_index_dimensions (ufl.classes.Cross attribute), 104
ufl_index_dimensions (ufl.classes.Curl attribute), 107
ufl_index_dimensions (ufl.classes.Determinant attribute), 104
ufl_index_dimensions (ufl.classes.Deviatoric attribute), 104
ufl_index_dimensions (ufl.classes.Div attribute), 107
ufl_index_dimensions (ufl.classes.Division attribute), 102
ufl_index_dimensions (ufl.classes.Dot attribute), 103
ufl_index_dimensions (ufl.classes.ExprList attribute), 105
ufl_index_dimensions (ufl.classes.ExprMapping attribute), 106
ufl_index_dimensions (ufl.classes.FacetAvg attribute), 111
ufl_index_dimensions (ufl.classes.Grad attribute), 106
ufl_index_dimensions (ufl.classes.Imag attribute), 103
ufl_index_dimensions (ufl.classes.Indexed attribute), 99
ufl_index_dimensions (ufl.classes.IndexSum attribute), 105
ufl_index_dimensions (ufl.classes.Inner attribute), 103
ufl_index_dimensions (ufl.classes.Inverse attribute), 104
ufl_index_dimensions (ufl.classes.Label attribute), 101
ufl_index_dimensions (ufl.classes.ListTensor attribute), 100
ufl_index_dimensions (ufl.classes.MathFunction attribute), 109
ufl_index_dimensions (ufl.classes.MaxValue attribute), 109
ufl_index_dimensions (ufl.classes.MinValue attribute), 109
ufl_index_dimensions (ufl.classes.MultiIndex attribute), 98
ufl_index_dimensions (ufl.classes.NablaDiv attribute), 107
ufl_index_dimensions (ufl.classes.NablaGrad attribute), 107
ufl_index_dimensions (ufl.classes.Outer attribute), 103
ufl_index_dimensions (ufl.classes.Power attribute), 102
ufl_index_dimensions (ufl.classes.Product attribute), 102
ufl_index_dimensions (ufl.classes.Real attribute), 103
ufl_index_dimensions (ufl.classes.RealValue attribute), 99
ufl_index_dimensions (ufl.classes.ReferenceCurl attribute), 107
ufl_index_dimensions (ufl.classes.ReferenceDiv attribute), 107
ufl_index_dimensions (ufl.classes.ReferenceGrad attribute), 106
ufl_index_dimensions (ufl.classes.ReferenceValue attribute), 111
ufl_index_dimensions (ufl.classes.Restricted attribute), 105
ufl_index_dimensions (ufl.classes.ScalarValue attribute), 98
ufl_index_dimensions (ufl.classes.Skew attribute), 104
ufl_index_dimensions (ufl.classes.Sum attribute), 102
ufl_index_dimensions (ufl.classes.Sym attribute), 105
ufl_index_dimensions (ufl.classes.Terminal attribute), 89
ufl_index_dimensions (ufl.classes.Trace attribute), 104
ufl_index_dimensions (ufl.classes.Transposed attribute), 103
ufl_index_dimensions (ufl.classes.Variable attribute), 101
ufl_index_dimensions (ufl.classes.VariableDerivative attribute), 106
ufl_index_dimensions (ufl.classes.Zero attribute), 98
ufl_index_dimensions (ufl.conditional.Condition attribute), 123
ufl_index_dimensions (ufl.conditional.Conditional attribute), 123
ufl_index_dimensions (ufl.conditional.MaxValue attribute), 124
ufl_index_dimensions (ufl.conditional.MinValue attribute), 124
ufl_index_dimensions (ufl.constantvalue.RealValue attribute), 125
ufl_index_dimensions (ufl.constantvalue.ScalarValue attribute), 126
ufl_index_dimensions (ufl.constantvalue.Zero attribute), 126
ufl_index_dimensions (ufl.differentiation.CoefficientDerivative attribute), 126
ufl_index_dimensions (ufl.differentiation.CoordinateDerivative attribute), 127
ufl_index_dimensions (ufl.differentiation.Curl attribute), 127
ufl_index_dimensions (ufl.differentiation.Div attribute), 127
ufl_index_dimensions (ufl.differentiation.Grad attribute), 127
ufl_index_dimensions (ufl.differentiation.NablaDiv attribute), 127
ufl_index_dimensions (ufl.differentiation.NablaGrad attribute), 127
ufl_index_dimensions (ufl.differentiation.ReferenceCurl attribute), 128

ufl_index_dimensions (ufl.differentiation.ReferenceDiv attribute), 128
ufl_index_dimensions (ufl.differentiation.ReferenceGrad attribute), 128
ufl_index_dimensions (ufl.differentiation.VariableDerivative attribute), 128
ufl_index_dimensions (ufl.exprcontainers.ExprList attribute), 130
ufl_index_dimensions (ufl.exprcontainers.ExprMapping attribute), 130
ufl_index_dimensions (ufl.indexed.Indexed attribute), 143
ufl_index_dimensions (ufl.indexsum.IndexSum attribute), 143
ufl_index_dimensions (ufl.mathfunctions.Atan2 attribute), 146
ufl_index_dimensions (ufl.mathfunctions.BesselFunction attribute), 146
ufl_index_dimensions (ufl.mathfunctions.MathFunction attribute), 146
ufl_index_dimensions (ufl.referencevalue.ReferenceValue attribute), 154
ufl_index_dimensions (ufl.restriction.Restricted attribute), 154
ufl_index_dimensions (ufl.tensoralgebra.Cofactor attribute), 155
ufl_index_dimensions (ufl.tensoralgebra.Cross attribute), 155
ufl_index_dimensions (ufl.tensoralgebra.Determinant attribute), 155
ufl_index_dimensions (ufl.tensoralgebra.Deviatoric attribute), 155
ufl_index_dimensions (ufl.tensoralgebra.Dot attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Inner attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Inverse attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Outer attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Skew attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Sym attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Trace attribute), 156
ufl_index_dimensions (ufl.tensoralgebra.Transposed attribute), 156
ufl_index_dimensions (ufl.tensors.ComponentTensor attribute), 157
ufl_index_dimensions (ufl.tensors.ListTensor attribute), 157
ufl_index_dimensions (ufl.variable.Label attribute), 158
ufl_index_dimensions (ufl.variable.Variable attribute), 159
ufl_mesh() (ufl.classes.MeshView method), 117
ufl_mesh() (ufl.domain.MeshView method), 129
ufl_mesh() (ufl.MeshView method), 165
ufl_operands (ufl.classes.Operator attribute), 97
ufl_operands (ufl.classes.Terminal attribute), 89
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), 84
ufl_shape (ufl.algebra.Product attribute), 84
ufl_shape (ufl.algebra.Real attribute), 85
ufl_shape (ufl.algebra.Sum attribute), 85
ufl_shape (ufl.Argument attribute), 172
ufl_shape (ufl.argument.Argument attribute), 85
ufl_shape (ufl.averaging.CellAvg attribute), 86
ufl_shape (ufl.averaging.FacetAvg attribute), 86
ufl_shape (ufl.CellNormal attribute), 166
ufl_shape (ufl.classes.Abs attribute), 102
ufl_shape (ufl.classes.Argument attribute), 100
ufl_shape (ufl.classes.Atan2 attribute), 110
ufl_shape (ufl.classes.BesselFunction attribute), 110
ufl_shape (ufl.classes.CellAvg attribute), 111
ufl_shape (ufl.classes.CellCoordinate attribute), 90
ufl_shape (ufl.classes.CellEdgeVectors attribute), 93
ufl_shape (ufl.classes.CellFacetJacobian attribute), 92
ufl_shape (ufl.classes.CellFacetJacobianInverse attribute), 95
ufl_shape (ufl.classes.CellFacetOrigin attribute), 91
ufl_shape (ufl.classes.CellNormal attribute), 95
ufl_shape (ufl.classes.CellOrigin attribute), 91
ufl_shape (ufl.classes.CellVertices attribute), 93
ufl_shape (ufl.classes.Coefficient attribute), 101
ufl_shape (ufl.classes.CoefficientDerivative attribute), 106
ufl_shape (ufl.classes.Cofactor attribute), 104
ufl_shape (ufl.classes.ComponentTensor attribute), 100
ufl_shape (ufl.classes.Condition attribute), 107
ufl_shape (ufl.classes.Conditional attribute), 108
ufl_shape (ufl.classes.Conj attribute), 103
ufl_shape (ufl.classes.CoordinateDerivative attribute), 106
ufl_shape (ufl.classes.Cross attribute), 104
ufl_shape (ufl.classes.Curl attribute), 107
ufl_shape (ufl.classes.Determinant attribute), 104
ufl_shape (ufl.classes.Deviatoric attribute), 104
ufl_shape (ufl.classes.Div attribute), 107
ufl_shape (ufl.classes.Division attribute), 102
ufl_shape (ufl.classes.Dot attribute), 104
ufl_shape (ufl.classes.ExprList attribute), 105
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), 92
ufl_shape (ufl.classes.FacetJacobianInverse attribute), 95
ufl_shape (ufl.classes.FacetNormal attribute), 95
ufl_shape (ufl.classes.FacetOrigin attribute), 91
ufl_shape (ufl.classes.GeometricQuantity attribute), 90
ufl_shape (ufl.classes.Grad attribute), 106
ufl_shape (ufl.classes.Identity attribute), 99
ufl_shape (ufl.classes.Imag attribute), 103
ufl_shape (ufl.classes.Indexed attribute), 99
ufl_shape (ufl.classes.IndexSum attribute), 105
ufl_shape (ufl.classes.Inner attribute), 103
ufl_shape (ufl.classes.Inverse attribute), 104
ufl_shape (ufl.classes.Jacobian attribute), 92
ufl_shape (ufl.classes.JacobianInverse attribute), 94
ufl_shape (ufl.classes.Label attribute), 101
ufl_shape (ufl.classes.ListTensor attribute), 100
ufl_shape (ufl.classes.MathFunction attribute), 109
ufl_shape (ufl.classes.MaxValue attribute), 109
ufl_shape (ufl.classes.MinValue attribute), 109
ufl_shape (ufl.classes.MultiIndex attribute), 98
ufl_shape (ufl.classes.NablaDiv attribute), 107
ufl_shape (ufl.classes.NablaGrad attribute), 107
ufl_shape (ufl.classes.Outer attribute), 103
ufl_shape (ufl.classes.PermutationSymbol attribute), 99
ufl_shape (ufl.classes.Power attribute), 102
ufl_shape (ufl.classes.Product attribute), 102
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), 107
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), 111
ufl_shape (ufl.classes.Restricted attribute), 105
ufl_shape (ufl.classes.ScalarValue attribute), 98
ufl_shape (ufl.classes.Skew attribute), 104
ufl_shape (ufl.classes.SpatialCoordinate attribute), 90
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), 103
ufl_shape (ufl.classes.Variable attribute), 101
ufl_shape (ufl.classes.VariableDerivative attribute), 106
ufl_shape (ufl.classes.Zero attribute), 98
ufl_shape (ufl.Coefficient attribute), 173
ufl_shape (ufl.coefficient.Coefficient attribute), 122
ufl_shape (ufl.conditional.Condition attribute), 123
ufl_shape (ufl.conditional.Conditional attribute), 123
ufl_shape (ufl.conditional.MaxValue attribute), 124
ufl_shape (ufl.conditional.MinValue attribute), 124
ufl_shape (ufl.constantvalue.Identity attribute), 125
ufl_shape (ufl.constantvalue.PermutationSymbol attribute), 125
ufl_shape (ufl.constantvalue.RealValue attribute), 125
ufl_shape (ufl.constantvalue.ScalarValue attribute), 126
ufl_shape (ufl.constantvalue.Zero attribute), 126
ufl_shape (ufl.differentiation.CoefficientDerivative attribute), 126
ufl_shape (ufl.differentiation.CoordinateDerivative attribute), 127
ufl_shape (ufl.differentiation.Curl attribute), 127
ufl_shape (ufl.differentiation.Div attribute), 127
ufl_shape (ufl.differentiation.Grad attribute), 127
ufl_shape (ufl.differentiation.NablaDiv attribute), 127
ufl_shape (ufl.differentiation.NablaGrad attribute), 127
ufl_shape (ufl.differentiation.ReferenceCurl attribute), 128
ufl_shape (ufl.differentiation.ReferenceDiv attribute), 128
ufl_shape (ufl.differentiation.ReferenceGrad attribute), 128
ufl_shape (ufl.differentiation.VariableDerivative attribute), 128
ufl_shape (ufl.exprcontainers.ExprList attribute), 130
ufl_shape (ufl.exprcontainers.ExprMapping attribute), 130
ufl_shape (ufl.FacetNormal attribute), 166
ufl_shape (ufl.geometry.CellCoordinate attribute), 135
ufl_shape (ufl.geometry.CellEdgeVectors attribute), 135
ufl_shape (ufl.geometry.CellFacetJacobian attribute), 135
ufl_shape (ufl.geometry.CellFacetJacobianInverse attribute), 136
ufl_shape (ufl.geometry.CellFacetOrigin attribute), 136
ufl_shape (ufl.geometry.CellNormal attribute), 136
ufl_shape (ufl.geometry.CellOrigin attribute), 137
ufl_shape (ufl.geometry.CellVertices attribute), 137
ufl_shape (ufl.geometry.FacetCoordinate attribute), 138
ufl_shape (ufl.geometry.FacetEdgeVectors attribute), 138
ufl_shape (ufl.geometry.FacetJacobian attribute), 138
ufl_shape (ufl.geometry.FacetJacobianInverse attribute), 139
ufl_shape (ufl.geometry.FacetNormal attribute), 139
ufl_shape (ufl.geometry.FacetOrigin attribute), 139
ufl_shape (ufl.geometry.GeometricQuantity attribute), 140
ufl_shape (ufl.geometry.Jacobian attribute), 140
ufl_shape (ufl.geometry.JacobianInverse attribute), 140
ufl_shape (ufl.geometry.ReferenceCellEdgeVectors attribute), 141
ufl_shape (ufl.geometry.ReferenceFacetEdgeVectors attribute), 141
ufl_shape (ufl.geometry.ReferenceNormal attribute), 142
ufl_shape (ufl.geometry.SpatialCoordinate attribute), 142
ufl_shape (ufl.Identity attribute), 174

ufl_shape (ufl.indexed.Indexed attribute), 143
ufl_shape (ufl.indexsum.IndexSum attribute), 143
ufl_shape (ufl.Jacobian attribute), 166
ufl_shape (ufl.JacobianInverse attribute), 167
ufl_shape (ufl.mathfunctions.Atan2 attribute), 146
ufl_shape (ufl.mathfunctions.BesselFunction attribute), 146
ufl_shape (ufl.mathfunctions.MathFunction attribute), 147
ufl_shape (ufl.PermutationSymbol attribute), 174
ufl_shape (ufl.referencevalue.ReferenceValue attribute), 154
ufl_shape (ufl.restriction.Restricted attribute), 154
ufl_shape (ufl.SpatialCoordinate attribute), 165
ufl_shape (ufl.tensoralgebra.Cofactor attribute), 155
ufl_shape (ufl.tensoralgebra.Cross attribute), 155
ufl_shape (ufl.tensoralgebra.Determinant attribute), 155
ufl_shape (ufl.tensoralgebra.Deviatoric attribute), 155
ufl_shape (ufl.tensoralgebra.Dot attribute), 156
ufl_shape (ufl.tensoralgebra.Inner attribute), 156
ufl_shape (ufl.tensoralgebra.Inverse attribute), 156
ufl_shape (ufl.tensoralgebra.Outer attribute), 156
ufl_shape (ufl.tensoralgebra.Skew attribute), 156
ufl_shape (ufl.tensoralgebra.Sym attribute), 156
ufl_shape (ufl.tensoralgebra.Trace attribute), 156
ufl_shape (ufl.tensoralgebra.Transposed attribute), 156
ufl_shape (ufl.tensors.ComponentTensor attribute), 157
ufl_shape (ufl.tensors.ListTensor attribute), 157
ufl_shape (ufl.variable.Label attribute), 159
ufl_shape (ufl.variable.Variable attribute), 159
ufl_sub_spaces() (ufl.classes.AbstractFunctionSpace method), 118
ufl_sub_spaces() (ufl.classes.FunctionSpace method), 118
ufl_sub_spaces() (ufl.classes.MixedFunctionSpace method), 118
ufl_sub_spaces() (ufl.classes.TensorProductFunctionSpace method), 118
ufl_sub_spaces() (ufl.FunctionSpace method), 172
ufl_sub_spaces() (ufl.functionspace.AbstractFunctionSpace method), 134
ufl_sub_spaces() (ufl.functionspace.FunctionSpace method), 134
ufl_sub_spaces() (ufl.functionspace.MixedFunctionSpace method), 134
ufl_sub_spaces() (ufl.functionspace.TensorProductFunctionSpace method), 134
UFLException, 163
undefined() (ufl.algorithms.MultiFunction method), 75
undefined() (ufl.algorithms.Transformer method), 74
undefined() (ufl.algorithms.transformer.Transformer method), 72
unexpected() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
unique_sorted_indices() (in module ufl.index_combination_utils), 143
unique_tuple() (in module ufl.algorithms.analysis), 47
unit_indexed_tensor() (in module ufl.tensors), 158
unit_list() (in module ufl.tensors), 158
unit_list2() (in module ufl.tensors), 158
unit_matrices() (in module ufl), 174
unit_matrices() (in module ufl.tensors), 158
unit_matrix() (in module ufl), 174
unit_matrix() (in module ufl.tensors), 158
unit_vector() (in module ufl), 174
unit_vector() (in module ufl.tensors), 158
unit_vectors() (in module ufl), 174
unit_vectors() (in module ufl.tensors), 158
unwrap_list_tensor() (in module ufl.tensors), 158

V

validate_form() (in module ufl.algorithms), 76
validate_form() (in module ufl.algorithms.checks), 59
value() (ufl.classes.ScalarValue method), 98
value() (ufl.constantvalue.ScalarValue method), 126
value_shape() (ufl.classes.FiniteElementBase method), 113
value_shape() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
value_shape() (ufl.FiniteElementBase method), 168
value_size() (ufl.classes.FiniteElementBase method), 113
value_size() (ufl.finiteelement.finiteelementbase.FiniteElementBase method), 80
value_size() (ufl.FiniteElementBase method), 168
Variable (class in ufl.classes), 101
Variable (class in ufl.variable), 159
variable() (in module ufl), 177
variable() (in module ufl.operators), 152
variable() (ufl.algorithms.apply_derivatives.GenericDerivativeRuleset method), 51
variable() (ufl.algorithms.apply_derivatives.VariableRuleset method), 52
variable() (ufl.algorithms.apply_restrictions.RestrictionPropagator method), 56
variable() (ufl.algorithms.check_arities.ArityChecker method), 58
variable() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
variable() (ufl.algorithms.formtransformations.PartExtracter method), 68
variable() (ufl.algorithms.renumbering.VariableRenumberingTransformer method), 70
variable() (ufl.algorithms.ReuseTransformer method), 74
variable() (ufl.algorithms.transformer.CopyTransformer method), 71
variable() (ufl.algorithms.transformer.ReuseTransformer method), 72

variable() (ufl.algorithms.transformer.VariableStripper method), 72
variable_derivative() (ufl.algorithms.apply_derivatives.DerivativeRuleDispatcher method), 49
variable_derivative() (ufl.algorithms.estimate_degrees.SumDegreeEstimator method), 65
VariableDerivative (class in ufl.classes), 106
VariableDerivative (class in ufl.differentiation), 128
VariableRenumberingTransformer (class in ufl.algorithms.renumbering), 70
VariableRuleset (class in ufl.algorithms.apply_derivatives), 52
VariableStripper (class in ufl.algorithms.transformer), 72
variant() (ufl.classes.FiniteElement method), 113
variant() (ufl.FiniteElement method), 168
variant() (ufl.finiteelement.finiteelement.FiniteElement method), 79
VectorConstant() (in module ufl), 173
VectorConstant() (in module ufl.coefficient), 122
VectorElement (class in ufl), 169
VectorElement (class in ufl.classes), 114
VectorElement (class in ufl.finiteelement.mixedelement), 82
visit() (ufl.algorithms.Transformer method), 74
visit() (ufl.algorithms.transformer.Transformer method), 72

W

warning() (ufl.log.Logger method), 145
warning_blue() (ufl.log.Logger method), 145
warning_green() (ufl.log.Logger method), 145
warning_red() (ufl.log.Logger method), 145

X

x (ufl.algorithms.domain_analysis.ExprTupleKey attribute), 61

Z

Zero (class in ufl.classes), 98
Zero (class in ufl.constantvalue), 126
zero() (in module ufl), 174
zero() (in module ufl.constantvalue), 126
zero() (ufl.algorithms.expand_indices.IndexExpander method), 66
zero() (ufl.algorithms.renumbering.IndexRenumberingTransformer method), 70
zero_expr() (in module ufl.algorithms.formtransformations), 69
zero_lists() (in module ufl.formoperators), 134