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 `Argument`s 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 `Argument`s, 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 `Coefficient`s 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 `Argument`s and two `Coefficient`s 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 values 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

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

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

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

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

$$\begin{aligned} \mathbf{A} : \mathbf{B} &= A_{ij} B^*_{ij} \\ \mathcal{C} : \mathcal{D} &= C_{ijk} D^*_{ijk} \end{aligned}$$

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

$$\mathcal{C} \otimes \mathcal{D} = C^*_{\iota^a_0 \dots \iota^a_{r-1}} D_{\iota^b_0 \dots \iota^b_{s-1}} \mathbf{i}_{\iota^a_0} \otimes \cdots \otimes \mathbf{i}_{\iota^a_{r-2}} \otimes \mathbf{i}_{\iota^b_1} \otimes \cdots \otimes \mathbf{i}_{\iota^b_{s-1}}$$

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:

$$\begin{aligned} \mathbf{v} \otimes \mathbf{u} &= v^*_i u_j \mathbf{i}_i \mathbf{i}_j, \\ \mathbf{v} \otimes \mathbf{B} &= v^*_i B_{kl} \mathbf{i}_i \mathbf{i}_k \mathbf{i}_l, \\ \mathbf{A} \otimes \mathbf{B} &= A^*_{ij} B_{kl} \mathbf{i}_i \mathbf{i}_j \mathbf{i}_k \mathbf{i}_l. \end{aligned}$$

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:

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

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

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

`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

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

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

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

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

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

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

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:

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

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

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

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

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

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:

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

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 $|v|_A = v^T Av$ 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.