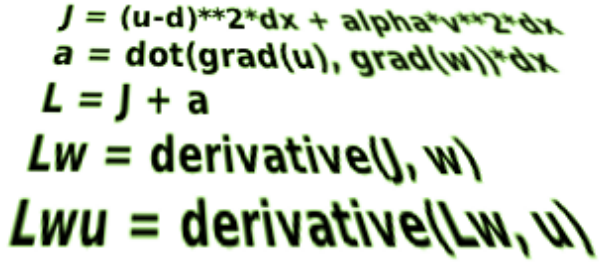
A domain specific language and symbolic library for FEM formulations of PDEs

The Unified Form Language – UFL



Martin Sandve Alnæs

Center for Biomedical Computing,
Simula Research Laboratory,
Oslo, Norway

June 18th, 2012
Imperial College, London

Overview of talk

- What the Unified Form Language is
- An overview of the language
- Some selected algorithm details
- Equation examples

Topics

What the Unified Form Language is

An overview of the language

Some selected algorithm details

Usage examples

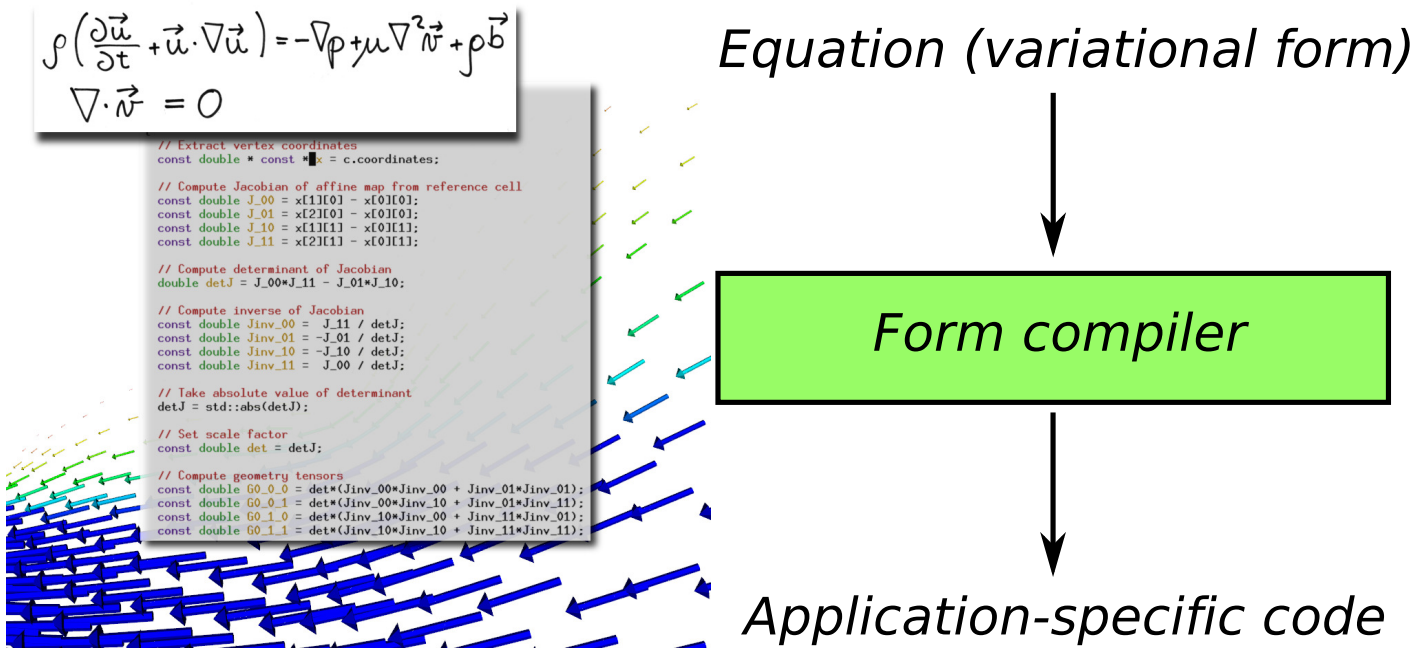
Automatic code generation principles

Input

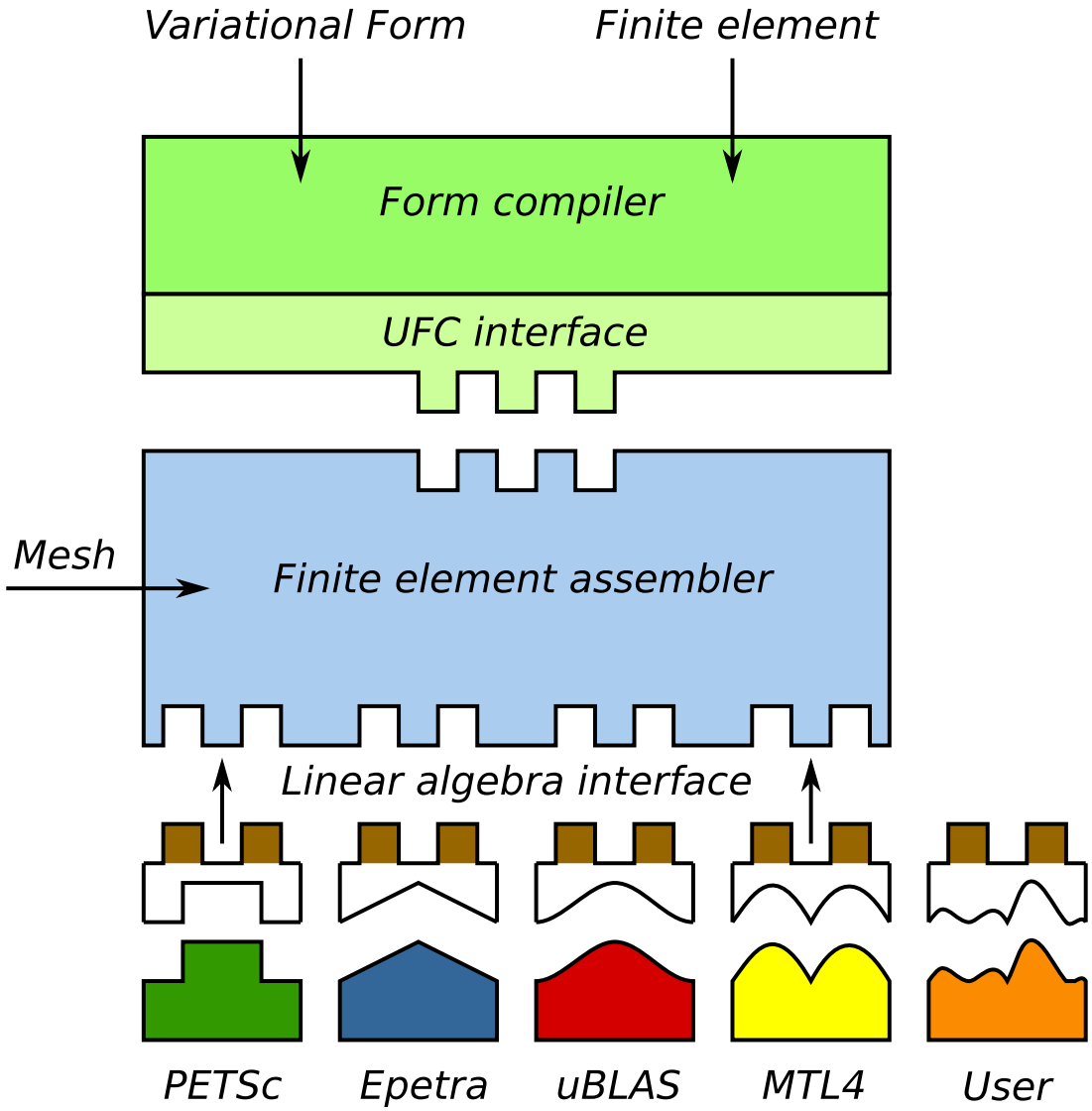
Equation (variational problem)

Output

Efficient application-specific code



Finite element assembly interfaces in FEniCS



UFL is a domain specific language (DSL) for PDEs

UFL allows the declaration (not computation) of:

- Choice of finite element spaces.
- Functions in these spaces.
- Expressions depending on functions and geometry.
- Integrals of expressions over subdomains.
- Transformations of forms, including differentiation.

UFL is a library for representation and manipulation of symbolic expressions

- Expressions are represented as expression trees
- The values and operators available are designed specifically for finite element methods, differential equations, and tensor algebra expressions
- Algorithms for differentiation and other symbolic manipulations are built in

UFL is a compiler frontend

UFL is part of the code generation pipeline in FEniCS:

- You write up your equations in UFL.
- The symbolic UFL representation of your equations is passed to a *form compiler*, which generates efficient low level code.
- This generated low level code is passed together with mesh and coefficient data to an *assembler* to assemble matrices, vectors, and scalars (from functionals).

Some other FEniCS components are FFC, UFC, DOLFIN.

Topics

What the Unified Form Language is

An overview of the language

Some selected algorithm details

Usage examples

A simple example equation

$$a(u, v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x, \tag{1}$$

$$L(v; f) = \int_\Omega f \, v \, \mathrm{d}x \tag{2}$$

```
1  cell = tetrahedron
2  V = FiniteElement("Lagrange", cell, 1)
3
4  f = Coefficient(V)
5  u = TrialFunction(V)
6  v = TestFunction(V)
7
8  a = dot(grad(u),grad(v))*dx
9  L = f*v*dx
```

Tree representation of right hand side

```
1  L = f*v*dx
2  print ufl.algorithms.tree_format(L)
```

```
1  Form:
2      Integral:
3          domain type: cell
4          domain id: 0
5          integrand:
6              Product
7              (
8                  Argument(FiniteElement(...), -2)
9                  Coefficient(FiniteElement(...), 0)
10             )
```

Tree representation of left hand side

```
1  a = dot(grad(u),grad(v))*dx
2  print ufl.algorithms.tree_format(a)
```

```
1  Form:
2      Integral:
3          domain type: cell
4          domain id: 0
5          integrand:
6              Dot
7              (
8                  Grad
9                      Argument(FiniteElement(...), -1)
10                 Grad
11                     Argument(FiniteElement(...), -2)
12             )
```

Functionals and variational forms are modelled as sums of integrals over subdomains

$$a(u, v; c) = \int_{\Omega_1} uv \, \mathrm{d}x + \int_{\Omega_2} c \, \mathrm{grad}\, u \cdot \mathrm{grad}\, v \, \mathrm{d}x, \tag{3}$$

$$L(v; f, g) = \int_{\Omega_1} fv \, \mathrm{d}x + \int_{\partial\Omega_2} gv \, \mathrm{d}s \tag{4}$$

```
1 a = u*v*dx(1) + c*dot(grad(u),grad(v))*dx(2)
2 L = f*v*dx(1) + g*v*ds(2)
```

A sublanguage for choosing finite element spaces

Finite elements can be declared as

```
1 U = FiniteElement(family, cell, degree)
2 V = VectorElement(family, cell, degree[, dim])
3 T = TensorElement(family, cell, degree[, shape[, symmetry]])
```

and combined into mixed element hierarchies:

```
1 TH = V*U;    H = (V*U)*(T*V);    M = MixedElement(T, V, U)
```

- Family is a string like "*Lagrange*", "*DG*", "*Nedelec*", etc.
- A cell is usually **interval**, **triangle**, **tetrahedron**.
- UFL does not know anything about a basis for these spaces.

Terminal expression types in UFL

- Literal constants (`3.14`, `Identity(3)`, `PermutationSymbol(3)`)
- Geometric quantities (`cell.x`, `cell.n`, `cell.volume`, etc.)
- Functions: Coefficient functions and argument functions

General properties of declared functions in UFL

- UFL allows the declaration of functions in specified function spaces

$$f \in V, \tag{5}$$

  which is (usually) a finite element space.

- Each function is a tensor valued function of $x \in \mathrm{R}^d$

$$f : \Omega \mapsto \mathrm{R}^s, \qquad s = (d_1, \ldots, d_r). \tag{6}$$

Coefficient functions represent functions given at assembly time

A coefficient function f represents a given discrete function

$$f(x) = \sum_i f_i \phi_i(x) \in V. \tag{7}$$

Example declaration:

```
1 V = VectorElement('CG', tetrahedron, 2)
2 f = Coefficient(V)
```

Note that the basis $\{\phi_i\}$ is typically provided by the form compiler, while the degrees of freedom $\{f_i\}$ are provided when the form is assembled.

Argument functions represent any function in a finite element space

```
1 u = TrialFunction(V) # an Argument
2 v = TestFunction(V)  # an Argument
3 f = Coefficient(V)
4 M = f**2*dx
5 L = f*v*dx
6 a = u*v*dx
```

Each `Argument` a form depends on adds to its arity, e.g.:

- The functional M depends on no arguments.
- The linear form L depends on one argument.
- The bilinear form a depends on two arguments.

A brief overview of available operators on expressions

- Basic arithmetic operators `+ - * / **`
- Scalar nonlinear functions **`sqrt`**, **`exp`**, **`ln`**, **`abs`**
- Scalar trigonometric functions
  `cos`, **`sin`**, **`tan`**, **`acos`**, **`asin`**, **`atan`**
- Product operators **`outer`**, **`inner`**, **`dot`**, **`cross`**
- Other common operators from tensor algebra are
  `transpose` `(A.T)`, **`tr`**, **`dev`**, **`skew`**, **`sym`**, **`det`**

Tensor algebra and index notation

Example using both tensor valued expressions and index notation

$$u : x \mapsto R^d, \qquad v : x \mapsto R^d, \qquad M : x \mapsto R^{d,d}. \tag{8}$$

$$a_1(u, v; M) = \int_\Omega (\operatorname{grad} u \cdot M) : \operatorname{grad} v \, \mathrm{d}x, \tag{9}$$

$$a_2(u, v; M) = \int_\Omega (M^T \nabla u) : \nabla v \, \mathrm{d}x, \tag{10}$$

$$a_3(u, v; M) = \int_\Omega M_{ij} u_{k,i} v_{k,j} \, \mathrm{d}x \tag{11}$$

```
1 a1 = inner(dot(grad(u), M), grad(v))*dx
2 a2 = inner(M.T*nabla_grad(u), nabla_grad(v))*dx
3 a3 = M[i,j] * u[k].dx(i) * v[k].dx(j) * dx
```

You can switch between tensor and index notation freely

$$A \quad | \quad A_{ijk} = u_i \frac{\mathrm{d}v_j}{\mathrm{d}x_k}, \tag{12}$$

$$B \quad | \quad B_{jki} = A_{ijk}. \tag{13}$$

```
1 Aijk = u[i] * v[j].dx(k)
2 A = as_tensor(Aijk, (i,j,k))
3 B = as_tensor(Aijk, (j,k,i))
```

Conditional operators allow simple branching

$$f = \begin{cases} g, & \text{if } \; c \\ h, & \text{otherwise.} \end{cases} \tag{14}$$

```
1 c = lt(abs(g), abs(h))
2 f = conditional(c, g, h)
```

Available boolean operators are named

eq, **ne**, **le**, **ge**, **lt**, **gt**, **And**, **Or**, **Not**.

Same as ternary operator in C (`c ? g: h`).

Operators to support Discontunuous Galerkin methods

- Restrict any expression to positive or negative side of a facet: `v('+'), v('-')`
- Operators **`jump`**`(v)` and **`avg`**`(v)`
- Integrate over set of interior facets Γ^k in mesh using `integrand*`**`dS`**`(k)`

Spatial differential operators

- $\frac{\mathrm{d}f}{\mathrm{d}x_i}$ can be written `f.dx(i)` or `Dx(f, i)`
- Common compound differential operators are
  `grad`, `div`, `nabla_grad`, `nabla_div`, `curl`, `rot`, `Dn`.

Derivatives w.r.t. user defined variables can be expressed with 'variable' and 'diff'

Say you want to express:

$$v = 3x^2, \tag{15}$$

$$f = f(v) = sin(v) + 3x^2, \tag{16}$$

$$g = \frac{\mathrm{d}f}{\mathrm{d}v} = cos(v). \tag{17}$$

In UFL this is done by annotating an expression as a **variable**:

```
1 v = 3*x**2
2 v = variable(v)
3 f = sin(v) + 3*x**2
4 g = diff(f, v)
```

If **diff** is applied to a form, it is applied to each integrand.

Some high level transformations of forms (1/2)

Consider the example bilinear form

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

With this you can

- Replace a coefficient function with another expression

  `replace(a, { f: g }) == dot(grad(g*u),grad(v))*dx`

- Construct the action of a bilinear form on a coefficient

  `action(a, g) == dot(grad(f*g),grad(v))*dx`

Some high level transformations of forms (2/2)

- Construct the adjoint(*) of a bilinear form

  ```
  adjoint(a) == dot(grad(f*u2),grad(v2))*dx
  ```

  where u2 and v2 are ordered opposite of u and v.
  (* only for cases where adjoint = transpose!)
- Compute the derivative of a form or functional

  ```
  derivative(a, u, du)
  ```

Topics

What the Unified Form Language is

An overview of the language

Some selected algorithm details

Usage examples

Some expression simplifications are carried out when constructing expression objects

Canonical ordering of sum and product terms:

- `a*b` $\rightarrow$ `a*b`, `b*a` $\rightarrow$ `a*b`

Simplification of identity and zero terms:

- `1*f` $\rightarrow$ `f`, `0*f` $\rightarrow$ `0`, `0+f` $\rightarrow$ `f`

Constant folding:

- `cos(0)` $\rightarrow$ `1`

Tensor component cancellations:

- `as_tensor(A[i,j], (i,j))` $\rightarrow$ `A`

Note how these simplifications work together with the differentiation chain rule:

- $\frac{d}{dx}(xf) = 1f + x0 \rightarrow f$.

Consider this example expression for explanation of the differentiation algorithm

With u a scalar (coefficient or argument) function,

$$u : x \mapsto \mathrm{R}, \tag{18}$$

consider the example expression

$$z = xe^{2u}, \tag{19}$$

and its derivative $z' \equiv \frac{\mathrm{d}z}{\mathrm{d}x}$

$$z' = e^{2u} + xe^{2u}u'. \tag{20}$$

We want to compute the symbolic representation of z' from z.

The symbolic representation of $z = xe^{2u}$ is an expression tree (or DAG) with vertices v_i

$$v_0 = 2, \quad \text{literal constant,} \tag{21}$$

$$v_1 = x, \quad \text{spatial coordinate,} \tag{22}$$

$$v_2 = u, \quad \text{coefficient function,} \tag{23}$$

(27)

The symbolic representation of $z = xe^{2u}$ is an expression tree (or DAG) with vertices v_i

$$
\begin{aligned}
v_0 &= 2, && \text{literal constant,} && (21)\\
v_1 &= x, && \text{spatial coordinate,} && (22)\\
v_2 &= u, && \text{coefficient function,} && (23)\\
v_3 &= v_0 v_2, && \text{product,} && (24)\\
v_4 &= e^{v_3}, && \exp, && (25)\\
v_5 &= v_1 v_4, && \text{product,} && (26)\\
& && && (27)
\end{aligned}
$$

The symbolic representation of $z = xe^{2u}$ is an expression tree (or DAG) with vertices v_i

$$v_0 = 2, \quad \text{literal constant,} \tag{21}$$
$$v_1 = x, \quad \text{spatial coordinate,} \tag{22}$$
$$v_2 = u, \quad \text{coefficient function,} \tag{23}$$
$$v_3 = v_0 v_2, \quad \text{product,} \tag{24}$$
$$v_4 = e^{v_3}, \quad \text{exp,} \tag{25}$$
$$v_5 = v_1 v_4, \quad \text{product,} \tag{26}$$
$$z \equiv v_5 = xe^{2u}. \tag{27}$$

The symbolic representation of $z' = \frac{d}{dx}(xe^{2u})$ is computed using the same algorithm underlying forward mode automatic differentiation

$$v_0 = 2, \qquad v_0' = 0, \tag{28}$$

$$v_1 = x, \qquad v_1' = 1, \tag{29}$$

$$v_2 = u, \qquad v_2' = u', \tag{30}$$

(34)

The symbolic representation of $z' = \frac{d}{dx}(xe^{2u})$ is computed using the same algorithm underlying forward mode automatic differentiation

$$v_0 = 2, \qquad v_0' = 0, \tag{28}$$

$$v_1 = x, \qquad v_1' = 1, \tag{29}$$

$$v_2 = u, \qquad v_2' = u', \tag{30}$$

$$v_3 = v_0 v_2, \qquad v_3' = v_0' v_2 + v_0 v_2', \tag{31}$$

$$v_4 = e^{v_3}, \qquad v_4' = v_4 v_3', \tag{32}$$

$$v_5 = v_1 v_4, \qquad v_5' = v_1' v_4 + v_1 v_4', \tag{33}$$

$$\tag{34}$$

The symbolic representation of $z' = \frac{d}{dx}(xe^{2u})$ is computed using the same algorithm underlying forward mode automatic differentiation

$$v_0 = 2, \qquad v_0' = 0, \tag{28}$$

$$v_1 = x, \qquad v_1' = 1, \tag{29}$$

$$v_2 = u, \qquad v_2' = u', \tag{30}$$

$$v_3 = v_0 v_2, \qquad v_3' = v_0' v_2 + v_0 v_2', \tag{31}$$

$$v_4 = e^{v_3}, \qquad v_4' = v_4 v_3', \tag{32}$$

$$v_5 = v_1 v_4, \qquad v_5' = v_1' v_4 + v_1 v_4', \tag{33}$$

$$z \equiv v_5, \qquad z' \equiv v_5' = e^{2u} + xe^{2u} 2u'. \tag{34}$$

Automatic functional differentiation explained

Preliminaries

Consider a functional F in functions g and h:

$$F(g,h) = \sum_r \int_{D_r} E_r(g,h)\,\mathrm{d}\mu_r. \tag{35}$$

For the purpose of explaining functional derivatives w.r.t. g, it is enough to consider

$$F(g) = \int_D E(g)\,\mathrm{d}\mu, \tag{36}$$

assuming $\frac{\mathrm{d}h}{\mathrm{d}g} = 0$.

Automatic functional differentiation explained

Definition

The Gateaux derivative of F w.r.t. $g \in V$ in a direction $\phi \in V$ is

$$D_{g,\phi}F(g) \equiv \frac{\mathrm{d}}{\mathrm{d}\tau}\left[F(g+\tau\phi)\right]_{\tau=0} = \int_D \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi)\right]_{\tau=0}, \quad (37)$$

assuming the domain D is independent of g.

Automatic functional differentiation explained

Basic differentiation rules

The computation of

$$D_{g,\phi}E(g) = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi)\right]_{\tau=0}. \tag{38}$$

requires differentiation rules for $D_{g,\phi}t$ for all types of terminal expression t.

$$D_{g,\phi}g = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[g+\tau\phi\right]_{\tau=0} = \phi, \tag{39}$$

(40)

Automatic functional differentiation explained

Basic differentiation rules

The computation of

$$D_{g,\phi}E(g) = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi)\right]_{\tau=0}. \tag{38}$$

requires differentiation rules for $D_{g,\phi}t$ for all types of terminal expression t.

$$D_{g,\phi}g = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[g+\tau\phi\right]_{\tau=0} = \phi, \tag{39}$$

$$D_{g,\phi}\frac{\mathrm{d}g}{\mathrm{d}x_i} = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[\frac{\mathrm{d}(g+\tau\phi)}{\mathrm{d}x_i}\right]_{\tau=0} = \frac{\mathrm{d}}{\mathrm{d}x_i}\left(\left[\frac{\mathrm{d}(g+\tau\phi)}{\mathrm{d}\tau}\right]_{\tau=0}\right) = \frac{\mathrm{d}\phi}{\mathrm{d}x_i}. \tag{40}$$

Automatic functional differentiation explained

Basic differentiation rules

The computation of

$$D_{g,\phi}E(g) = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi)\right]_{\tau=0}. \tag{38}$$

requires differentiation rules for $D_{g,\phi}t$ for all types of terminal expression t.

$$D_{g,\phi}g = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[g+\tau\phi\right]_{\tau=0} = \phi, \tag{39}$$

$$D_{g,\phi}\frac{\mathrm{d}g}{\mathrm{d}x_i} = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[\frac{\mathrm{d}(g+\tau\phi)}{\mathrm{d}x_i}\right]_{\tau=0} = \frac{\mathrm{d}}{\mathrm{d}x_i}\left(\left[\frac{\mathrm{d}(g+\tau\phi)}{\mathrm{d}\tau}\right]_{\tau=0}\right) = \frac{\mathrm{d}\phi}{\mathrm{d}x_i}. \tag{40}$$

For coefficient functions other than g and all other terminal values, $D_{g,\phi}t = 0$.

Derivatives w.r.t functions in mixed spaces are allowed

$$D_{(u,p),(\phi,\eta)}E(u,p) = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(u+\tau\phi, p+\tau\eta)\right]_{\tau=0}. \qquad (41)$$

```
1  V = VectorElement("CG", cell, 2)
2  P = FiniteElement("CG", cell, 1)
3  TH = V*P
4
5  u = Coefficient(V)
6  p = Coefficient(P)
7  dup = TestFunction(TH)
8
9  F = derivative(E(u,p)*dx, (u,p), dup)
```

Coefficient dependencies can be specified when differentiating

$$D_{g,\phi}E(g,h) = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi, h+\tau\frac{dh}{dg}\phi)\right]_{\tau=0}. \qquad (42)$$

```
1  P = FiniteElement("CG", cell, 1)
2
3  g = Coefficient(P) # Differentiation variable
4  dg = Argument(P)   # Variation direction
5  h = Coefficient(P) # Dependent coefficient
6  f = Coefficient(P) # dh/dg
7
8  # Yields (dg*h)*dx:
9  F1 = derivative(g*h*dx, g, dg)
10 # Yields (dg*h + g*f*dg)*dx:
11 F2 = derivative(g*h*dx, g, dg, coefficient_derivatives={ g: f })
```

Topics

What the Unified Form Language is

An overview of the language

Some selected algorithm details

Usage examples

Example uses of differentiation features

Typical uses of the differentiation features in UFL are

- Exact linearization of nonlinear residual equation for Newtons method
- Differentiation of material laws in e.g. hyperelastic equations.
- Differentiation of Lagrangian functional to form the optimality system in PDE constrained optimization.
- Computing a source term symbolically for validation of a solver.
- Sensitivity analysis.

Example: Stokes equations, taken from DOLFIN demo directory

```
1  P2 = VectorElement("Lagrange", tetrahedron, 2)
2  P1 = FiniteElement("Lagrange", tetrahedron, 1)
3  TH = P2 * P1
4
5  u, p = TrialFunctions(TH)
6  v, q = TestFunctions(TH)
7
8  f = Coefficient(P2)
9
10 a = (inner(grad(u), grad(v)) + div(v)*p + div(u)*q)*dx
11 L = dot(f, v)*dx
```

Computational hemodynamics

```
# Define Cauchy stress tensor
def sigma(v, w):
    I = Identity(v.cell().d)
    return 2.0*mu*0.5*(grad(v) + grad(v).T) - w*I

def epsilon(v):   # Define symmetric gradient
    return 0.5*(grad(v) + grad(v).T)

# Tentative velocity step (sigma formulation)
U = 0.5*(u0 + u)
F1 = (rho*(1/k)*inner(v, u - u0)*dx
   + rho*inner(v, grad(u0)*(u0 - w))*dx
   + inner(epsilon(v), sigma(U, p0))*dx
   + inner(v, p0*n)*ds - mu*inner(grad(U).T*n, v)*ds
   - inner(v, f)*dx)
a1, L1 = lhs(F1), rhs(F1)

# Pressure correction
a2 = inner(grad(q), k*grad(p))*dx
L2 = inner(grad(q), k*grad(p0))*dx - q*div(u1)*dx

# Velocity correction
a3 = inner(v, u)*dx
L3 = inner(v, u1)*dx + inner(v, k*grad(p0 - p1))*dx
```

Valen-Sendstad, Mardal, Logg, *Computational hemodynamics* (2011)

Example: Snippet from an optimal control code

```
# Define Lagrangian functional
def J(u, v):
    return 0.5*(u-d)**2*dx + 0.5*alpha*v**2*dx
def a(u, w):
    return dot(grad(u), grad(w))*dx
def b(v, w):
    return v*w*dx
L = J(u, v) + a(u, w) - b(v, w)

# Derive equations for u, then w
Lw = derivative(L, w); Lwu = derivative(Lw, u)
Lu = derivative(L, u); Luw = derivative(Lu, w)

# Alternatively derive full optimality system at once:
F = derivative(L, (u,v,w))
J = derivative(F, (u,v,w))
```

Example: Hyperelasticity equations(1/2), taken from DOLFIN demo directory

```
1  cell = tetrahedron
2  V = VectorElement("Lagrange", cell, 1)
3
4  du = TrialFunction(V)   # Incremental displacement
5  v  = TestFunction(V)    # Test function
6
7  u = Coefficient(V)      # Displacement from previous iteration
8  B = Coefficient(V)      # Body force per unit volume
9  T = Coefficient(V)      # Traction force on the boundary
10
11 # Elasticity parameters
12 mu    = Constant(cell)
13 lmbda = Constant(cell)
```

Example: Hyperelasticity equations(2/2), taken from DOLFIN demo directory

```
# Kinematics
I = Identity(cell.d)             # Identity tensor
F = I + grad(u)                  # Deformation gradient dX/dx
C = F.T*F                        # Right Cauchy-Green tensor
# Invariants of deformation tensors
Ic = tr(C); J = det(F)
# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2
# Total potential energy
Pi = psi*dx - inner(B, u)*dx - inner(T, u)*ds

# First variation of Pi (directional derivative
# about u in the direction of v)
F = derivative(Pi, u, v)
J = derivative(F, u, du)
```

Example: Hyperelasticity equations, Ogden type model (1/3)

```
1  cell = tetrahedron
2  d = cell.d
3  I = Identity(d)
4
5  V = VectorElement("CG", cell, 1)
6  Q = FiniteElement("DG", cell, 0)
7
8  u = Coefficient(V)
9
10 alpha1 = Constant(cell); mu1 = Constant(cell)
11 alpha2 = Constant(cell); mu2 = Constant(cell)
12 alpha3 = Constant(cell); mu3 = Constant(cell)
```

Example: Hyperelasticity equations, Ogden type model (2/3)

```
1  F = I + grad(u)
2  C = F*F.T
3  m = tr(C) / 3
4  CmI = C - m*I
5  q = det(CmI) / 2
6
7  p = inner(CmI, CmI.T) / 6
8  phi = atan(sqrt(p**3 - q**2) / q) / 3
9
10 l1 = m + 2*sqrt(p)*cos(phi)
11 l2 = m -   sqrt(p)*(cos(phi) + sqrt(3)*sin(phi))
12 l3 = m -   sqrt(p)*(cos(phi) - sqrt(3)*sin(phi))
```

Example: Hyperelasticity equations, Ogden type model (3/3)

```
1  psi = (  (mu1/alpha1) * (l1**alpha1 + l2**alpha1 + l3**alpha1)
2       + (mu2/alpha2) * (l1**alpha2 + l2**alpha2 + l3**alpha2)
3       + (mu3/alpha3) * (l1**alpha3 + l2**alpha3 + l3**alpha3) )
4  M = psi*dx
5
6  v = TestFunction(V)
7  du = TrialFunction(V)
8  F = derivative(M, u, v)
9  J = derivative(F, u, du)
```

Questions?

- All about the FEniCS project: *http://www.fenicsproject.org*
- The UFL project and source code: *http://www.launchpad.net/ufl*
- Ongoing work: a form compiler with a plugin mechanism for other FEM libraries. *http://www.launchpad.net/uflacs*
- *martinal@simula.no*