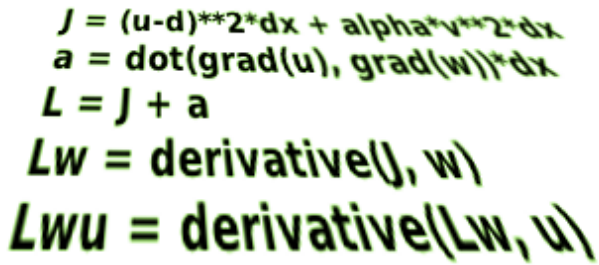
Automation of PDE constrained optimization

Algorithmic differentiation as abstract building blocks in high level algorithms



Martin Sandve Alnæs

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

September 12th
ECCOMAS 2012, Wien

Overview of talk

The algorithmic differentiation of variational forms as implemented in FEniCS (UFL) can be a powerful building block for high level optimization algorithms.

- Example PDE constrained optimization problem
- Automation of gradient based optimization algorithms
- Automation of Lagrangian based optimization algorithms
- Algorithmic differentiation in FEniCS explained

Example weak forms in the Unified Form Language showing tensor algebra and index notation

$$u : x \mapsto R^d, \qquad v : x \mapsto R^d, \qquad M : x \mapsto R^{d,d}. \tag{1}$$

$$a_1(u, v; M) = \int_\Omega (\operatorname{grad} u \cdot M) : \operatorname{grad} v \, \mathrm{d}x, \tag{2}$$

$$a_2(u, v; M) = \int_\Omega M_{ij} u_{k,i} v_{k,j} \, \mathrm{d}x \tag{3}$$

```
1 V = VectorElement("Lagrange", triangle, 3)
2 T = TensorElement("Lagrange", triangle, 1)
3 u = TrialFunction(V);  v = TestFunction(V);  M = Coefficient(T)
4 a1 = inner(dot(grad(u), M), grad(v))*dx
5 a2 = M[i,j] * u[k].dx(i) * v[k].dx(j) * dx
```

Example PDE constrained optimization problem

Minimize a cost functional J of the control $v \in V$ and state $u = u(v) \in U$

$$J(v) = \hat{J}(v, u(v)) = \int_{\Omega_D} (u - z)^2 \,\mathrm{d}x + \alpha \int_{\Omega_C} v^2 \,\mathrm{d}x, \tag{4}$$

constrained by the state equation (PDE)

$$a(u, w) = b(v; w), \qquad \forall w \in U, \tag{5}$$

$$u = 0, \qquad \text{on} \quad \partial\Omega. \tag{6}$$

For examples later I'll use

$$a(u, w) = \int_{\Omega} uw + \operatorname{grad} u \cdot \operatorname{grad} w \,\mathrm{d}x, \tag{7}$$

$$b(v; w) = \int_{\Omega} vw \,\mathrm{d}x. \tag{8}$$

V, J, a, b, etc. must be part of a problem definition.

Example problem class

```
1  class Problem:
2      def __init__(self, n=128, alpha=1e-4,
3                   zexpr="1.0+3.0*x[0]+exp(x[1])",
4                   v0expr="0.0"):
5          self.mesh = UnitSquare(128, 128)
6          self.V = FunctionSpace(self.mesh, "Lagrange", 1)
7          self.z = Function(self.V)
8          self.z.interpolate(Expression(zexpr))
9          # ... some more lines
10     def J(self, v, u):
11         return 0.5*(u-self.z)**2*dx + 0.5*self.alpha*v**2*dx
12     def a(self, u, w):
13         return (u*w + dot(grad(u), grad(w)))*dx
14     def b(self, v, w):
15         return v*w*dx
16     def a_adjoint(self, u, w):
17         return self.a(w, u)
```

Gradient based iterative methods require the computation of $D_{v,\eta}J(v)$ via duality arguments

$$D_{v,\eta}J(v) \equiv \frac{d}{d\tau}\left[J(v+\tau\eta)\right]_{\tau=0}, \qquad \forall \eta \in V_h, \tag{9}$$

which can be split into

$$D_{v,\eta}J = D_{v,\eta}\hat{J} + D_{u,\bar{u}}\hat{J}, \qquad \bar{u} \equiv D_{v,\eta}u(v). \tag{10}$$

First solve the dual equation for w, where $a^*(u,v) \equiv a(v,u)$,

$$a^*(w,\psi) = D_{u,\psi}\hat{J}, \qquad \forall \psi, \tag{11}$$

$$w = 0, \qquad \text{on} \quad \partial\Omega. \tag{12}$$

Then

$$D_{u,\bar{u}}\hat{J} = a^*(w,\bar{u}) = a(\bar{u},w) = b(\eta;w) \tag{13}$$

Example computation of gradient via adjoint equation

```
1  # Callback for scipy.optimize.fmin_l_bfgs_b
2  def func(x):
3      global p, u, v, w, phi, psi
4      v.vector()[:] = x
5      solve(p.a(phi, psi) == p.b(v, psi), u, p.bcu)
6
7      J = p.J(v, u)
8      dJdu = derivative(J, u, psi)
9      dJdv = derivative(J, v, psi)
10     Jvalue = assemble(J)
11
12     solve(p.a_adjoint(phi, psi) == dJdu, w, p.bcw)
13     DJ = assemble(dJdv + p.b(psi, w))
14
15     return Jvalue, DJ.array().copy()
```

One shot methods can be automated through differentiation of the Lagrangian functional

Define the Lagrangian functional

$$L(v, w, u) = J(v, u) + a(u, w) - b(v, w), \tag{14}$$

and differentiate it to find the optimality conditions for (v, w, u),

$$L_v = D_{v,\hat{v}} L(u, v, w) = 0, \qquad \forall \hat{v} \in V, \tag{15}$$

$$L_w = D_{w,\hat{w}} L(u, v, w) = 0, \qquad \forall \hat{w} \in U, \tag{16}$$

$$L_u = D_{u,\hat{u}} L(u, v, w) = 0, \qquad \forall \hat{u} \in U. \tag{17}$$

Then differentiate again to build the block system

$$\begin{bmatrix} L_{vv} & L_{vw} & 0 \\ L_{wv} & 0 & L_{wu} \\ 0 & L_{uw} & L_{uu} \end{bmatrix} \begin{bmatrix} v \\ w \\ u \end{bmatrix} = \begin{bmatrix} \alpha M & B^* & 0 \\ B & 0 & A \\ 0 & A^* & M \end{bmatrix} \begin{bmatrix} v \\ w \\ u \end{bmatrix} = \begin{bmatrix} -L_v \\ -L_w \\ -L_u \end{bmatrix} \tag{18}$$

Automation of one shot method preconditioning

Define the preconditioning norm

$$P(v, w, u) = \|v\|^2 + \|w\|^2 + \|u\|^2 + a(u, u) + a(w, w) \tag{19}$$

and differentiate it twice to find the block preconditioner system

$$\begin{bmatrix} M & 0 & 0 \\ 0 & M + A & 0 \\ 0 & 0 & M + A \end{bmatrix} \tag{20}$$

Choice of preconditioner not obvious, more work needed.

Example one shot solver

```
1  M = MixedFunctionSpace([p.V, p.V, p.V])
2  uvw = Function(M); u, v, w = split(uvw)
3  bcs = [DirichletBC(M.sub(0), 0, DomainBoundary()),
4         DirichletBC(M.sub(2), 0, DomainBoundary())]
5
6  L = p.J(v, u) + p.a(u, w) - p.b(v, w)
7  precnorm = 0.5*(u**2 + v**2 + w**2)*dx + p.a(u,u) + p.a(w,w)
8
9  F = derivative(L, uvw)
10 A, b = assemble_system(derivative(F, uvw), -F, bcs)
11 Pform = derivative(derivative(precnorm, uvw))
12 P, _ = assemble_system(Pform, -F, bcs)
13
14 solver = KrylovSolver("tfqmr", "amg")
15 solver.set_operators(A, P)
16 solver.solve(uvw.vector(), b)
```

Automatic functional differentiation is (almost) just differentiation of expressions

With no loss of generality w.r.t. multiple integrals or additional independent coefficients, we can consider a functional

$$F(g) = \int_D E(g)\,\mathrm{d}\mu. \tag{21}$$

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

$$D_{g,\phi}F(g) \equiv \frac{\mathrm{d}}{\mathrm{d}\tau}\left[F(g+\tau\phi)\right]_{\tau=0} = \int_D \frac{\mathrm{d}}{\mathrm{d}\tau}\left[E(g+\tau\phi)\right]_{\tau=0}, \tag{22}$$

assuming the domain D is independent of g.

Algorithmic differentiation of an expression tree is the chain rule plus differentiation rules for each type and operator

- Algorithm structure equivalent to forward mode AD.
- The innermost derivatives are computed first, recursively.
- Function gradients still represented after differentiation.

$$\operatorname{grad}(v * u) = \operatorname{grad} v * u + v * \operatorname{grad} u \tag{23}$$

Directional derivatives w.r.t. functions requires differentiation rules for $D_{g,\phi}t$ for all types of terminal expression t

Assuming coefficient functions g, h, we have

$$D_{g,\phi}g = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[g + \tau\phi\right]_{\tau=0} = \phi, \tag{24}$$

$$D_{g,\phi}\nabla g = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[\nabla(g + \tau\phi)\right]_{\tau=0} = \nabla\phi, \tag{25}$$

$$D_{g,\phi}\nabla h = \frac{\partial h}{\partial g}\phi. \tag{26}$$

The user can provide $\frac{\partial h}{\partial g}$, which by default is 0.

Algorithms for each differentiation variable type differ only by the terminal differentiation rules

```
1  V = FiniteElement("Lagrange", triangle, 1)
2  u = Coefficient(V)
3  w = TestFunction(V)
4  v = variable(u)
5  f = diff(v**2, v)                  # == 2*v
6  g = derivative(u**2*dx, u, w)      # == 2*u*w
```

Or considering nested differentiation,

$$\text{grad}\,(vu)] = \text{grad}\,(v)u + v\text{grad}\,(u), \tag{27}$$

$$f = \frac{d}{dv}[\text{grad}\,(vu)] = \text{grad}\,u, \tag{28}$$

$$g = D_{u,w}[\text{grad}\,(vu)] = \text{grad}\,(v)w + v\text{grad}\,(w). \tag{29}$$

Ways to use differentiation features of UFL

- Computing cost functional gradient.
- Differentiation of Lagrangian functional.

- Sensitivity analysis or parameter estimation.
- Exact linearization of nonlinear residual equation.
- Differentiation of e.g. hyperelasticity material laws.
- Computing a source term for validation of a solver.

Thank you!

Software links:

- *http://www.fenicsproject.org*
- *http://www.launchpad.net/ufl*
- *http://launchpad.net/cbc.block*
- *http://www.dolfin-adjoint.org*

Preconditioning papers:

- J. Schöberl and W. Zulehner, SJMAEL (2010)
- B.F. Nielsen and K.-A. Mardal, SISC (2012)

Questions:

- *https://answers.launchpad.net/fenics*
- *martinal@simula.no*

A simple example equation

$$a(u, v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x, \tag{30}$$

$$L(v; f, g) = \int_\Omega f \, v \, \mathrm{d}x + \int_{\partial\Omega} g \, v \, \mathrm{d}s \tag{31}$$

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

Tree representation of the weak Laplace form

```
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             )
```

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:

- $\frac{d}{dx}(xy) = 1y + x0 \rightarrow y + 0 \rightarrow y$.