The high-level algorithmic differentiation
of the FEniCS finite element system

P. E. Farrell, S. W. Funke, D. A. Ham, M. E. Rognes

Imperial College London
Simula Research Laboratory

June 10, 2013
Introduction
  High-level AD

User experience
  Symbolic representation of FEM
  The forward model
  Differentiation
  Optimisation

How it works
  Operator overloading
  Symbolic calculus

Discussion
  Performance
The dream

Properties desired of AD

- Fully automatic
- Extremely efficient \((R \leq 2\) for store-all)
- Works in parallel with no intervention
The dream

Properties desired of AD

- Fully automatic
- Extremely efficient ($R \leq 2$ for store-all)
- Works in parallel with no intervention

Bad news, good news

- Very hard in general
- Achievable via specialisation
The dream

Properties desired of AD

- Fully automatic
- Extremely efficient ($R \leq 2$ for store-all)
- Works in parallel with no intervention

Bad news, good news

- Very hard in general
- Achievable via specialisation (to finite elements)
High-level AD for finite elements

<table>
<thead>
<tr>
<th>“Low-level” AD</th>
<th>“High-level” AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>float, +, -, *, /</td>
<td>Function in a FunctionSpace</td>
</tr>
<tr>
<td>no specific math information</td>
<td>linear/nonlinear solves</td>
</tr>
<tr>
<td></td>
<td>symbolic structure available</td>
</tr>
</tbody>
</table>
High-level AD for finite elements

<table>
<thead>
<tr>
<th>&quot;Low-level&quot; AD</th>
<th>&quot;High-level&quot; AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Function in a FunctionSpace</td>
</tr>
<tr>
<td>+, -, *, /</td>
<td>linear/nonlinear solves</td>
</tr>
<tr>
<td>no specific math information</td>
<td>symbolic structure available</td>
</tr>
</tbody>
</table>

Similarities

- Operator overloading builds a DAG of information flow
- Propagate the chain rule forwards/backwards through graph
High-level AD for finite elements

<table>
<thead>
<tr>
<th>“Low-level” AD</th>
<th>“High-level” AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Function in a FunctionSpace</td>
</tr>
<tr>
<td>+, -, *, /</td>
<td>linear/nonlinear solves</td>
</tr>
<tr>
<td>no specific math information</td>
<td><strong>symbolic structure available</strong></td>
</tr>
</tbody>
</table>

Introspection

- Can inspect equation solves for dependencies
- Can manipulate equation terms (linearise/transpose)
High-level AD for finite elements

<table>
<thead>
<tr>
<th>“Low-level” AD</th>
<th>“High-level” AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Function in a FunctionSpace</td>
</tr>
<tr>
<td>+, −, *, /</td>
<td>linear/nonlinear solves</td>
</tr>
<tr>
<td>no specific math information</td>
<td>symbolic structure available</td>
</tr>
</tbody>
</table>

Performance

- Can exploit structure of equations for efficiency
- Two-phase linearisation, factorisation caching, ...
Symbolic representation in UFL

Burgers’ equation (strong)

\[ F = (u \cdot \nabla) u - \nu \nabla^2 u - f = 0 \]
### Symbolic representation in UFL

#### Burgers’ equation (strong)

\[ F = (u \cdot \nabla) u - \nu \nabla^2 u - f = 0 \]

#### Burgers’ equation (weak)

\[ F = ((u \cdot \nabla) u, v) + \nu (\nabla u, \nabla v) - (f, v) = 0 \]
### Symbolic representation in UFL

#### Burgers’ equation (strong)

\[ F = (u \cdot \nabla) u - \nu \nabla^2 u - f = 0 \]

#### Burgers’ equation (weak)

\[ F = ((u \cdot \nabla) u, v) + \nu (\nabla u, \nabla v) - (f, v) = 0 \]

#### Burgers’ equation (UFL code)

\[ F = (\text{grad}(u)*u*v + \nu*\text{grad}(u)*\text{grad}(v) - f*v)*dx == 0 \]
The forward model

```python
from dolfin import *

mesh = Mesh("some_mesh.xml")
V = VectorFunctionSpace(mesh, "CG", 2)

u = Function(V, "some_ic.xml"); u_next = Function(V)
v = TestFunction(V)

nu = Constant(0.0001)
timestep = Constant(0.01)

F = (inner((u_next - u)/timestep, v)
    + inner(grad(u_next)*u_next, v)
    + nu*inner(grad(u_next), grad(v)))*dx

bc = DirichletBC(V, (0.0, 0.0), "on_boundary")

t = 0.0
end = 0.1
while (t <= end):
    solve(F == 0, u_next, bc)
    u.assign(u_next)
    t += float(timestep)
```
from dolfin import *
from dolfin_adjoint import *

...

while (t <= end):
    solve(F == 0, u_next, bc)
    u.assign(u_next)
    t += float(timestep)

J = Functional(inner(u, u)*dx*dt[FINISH_TIME])
m = InitialConditionParameter(u)
dJdm = compute_gradient(J, m)
from dolfin import *
from dolfin_adjoint import *

... 

z = interpolate(Expression(('1.0', '0.0')), V)
J = Functional(inner(u-z, u-z)*dx*dt[FINISH_TIME])
m = InitialConditionParameter(u)
Jhat = ReducedFunctional(J, m)
m_opt = minimize(Jhat, method='L-BFGS-B')
The optimised model

```python
from dolfin import *
from dolfin_adjoint import *

...

z = interpolate(Expression(("1.0", "0.0")), V)
J = Functional(inner(u-z, u-z)*dx*dt[FINISH_TIME])
m = InitialConditionParameter(u)
Jhat = ReducedFunctional(J, m)
m_opt = minimize(Jhat, method="Newton-CG")
```
Operator overloading

Fundamental operators

In dolfin-adjoint, the two fundamental operators are:

- **solve**: solves a linear/nonlinear equation.
- **assign**: assigns one `Function` to another.

These are overloaded to build the tape at runtime.
Operator overloading

Fundamental operators

In dolfin-adjoint, the two fundamental operators are:

- **solve**: solves a linear/nonlinear equation.
- **assign**: assigns one `Function` to another.

These are overloaded to build the tape at runtime.

```
assign u_{ic} \rightarrow \hat{u}_0 \rightarrow u_1
assign \hat{u}_1 \rightarrow PDE
assign u_1 \rightarrow \hat{u}_2 \rightarrow u_3
assign PDE \rightarrow assign
```
In dolfin-adjoint, the two fundamental operators are:

▶ **solve**: solves a linear/nonlinear equation.
▶ **assign**: assigns one Function to another.

These are overloaded to build the tape at runtime.
Operator overloading

Fundamental operators

In dolfin-adjoint, the two fundamental operators are:

- **solve**: solves a linear/nonlinear equation.
- **assign**: assigns one `Function` to another.

These are overloaded to build the tape at runtime.
Adjoint assembly

UFL retains **symbolic information** about PDEs being solved.

**Symbolic calculus**

UFL can symbolically

- derive entire sparse Jacobians: `derivative(F, u)`
- transpose forms: `adjoint(a)`

All ingredients for adjoint assembly available.
Optimisations

Having the high-level structure available allows for many automatic optimisations that are otherwise impossible/very difficult.
Two-phase linearisation

**Forward problem**

Solve $F(u, m) = 0$ (taking $N$ linear solves).

**Piggyback linearisation**

Differentiate through each of the $N$ iterations.
Two-phase linearisation

**Forward problem**

Solve \( F(u, m) = 0 \) (taking \( N \) linear solves).

**Two-phase linearisation**

Solve in one iteration

\[
\frac{\partial F}{\partial u} \dot{u} = - \frac{\partial F}{\partial m} \dot{m}.
\]
Two-phase linearisation of the $p$-Laplace equation

$p$-Laplace equation

$$-\nabla \cdot \left( (\epsilon^2 + \frac{1}{2}|\nabla u|^2)^{p-2/2} \nabla u \right) = f$$

Solution domain

Solution
Two-phase linearisation of the $p$-Laplace equation

Functional

$$J = \int_{\Omega} \gamma(u)^2 \, dx$$
Two-phase linearisation of the $p$-Laplace equation

**Functional**

$$ J = \int_\Omega \gamma(u)^2 \ dx $$

**Code**

```python
J = Functional(inner(gamma(u), gamma(u))*dx)
m = TimeConstantParameter(f)
dJdm = compute_gradient(J, m)
```
Two-phase linearisation of the $p$-Laplace equation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (s)</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward model</td>
<td>2949.8</td>
<td></td>
</tr>
<tr>
<td>Piggyback</td>
<td>2890.7</td>
<td>1.9799</td>
</tr>
<tr>
<td>Two-phase</td>
<td>14.3</td>
<td><strong>1.0048</strong></td>
</tr>
</tbody>
</table>
Computing the Hessian

Higher derivatives

(Number of equations to solve) $\propto 2^{(\text{derivative order})}$

<table>
<thead>
<tr>
<th>to compute</th>
<th>you need</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{J}(m)$</td>
<td>$u$</td>
</tr>
<tr>
<td>$\frac{d\hat{J}}{dm}$</td>
<td>$u, \lambda$</td>
</tr>
<tr>
<td>$\frac{d^2\hat{J}}{dm^2}\delta m$</td>
<td>$u, \dot{u}, \lambda, \dot{\lambda}$</td>
</tr>
</tbody>
</table>
Equation structure

\[ F(u, m) = 0 \]

\[ \frac{\partial F}{\partial u} \lambda = \frac{\partial J}{\partial u} \]

\[ \frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m \]

\[ \frac{\partial F}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right) + \cdots \]
### Equation structure

<table>
<thead>
<tr>
<th>$F(u, m) = 0$</th>
<th>$\frac{\partial F^<em>}{\partial u} \lambda = \frac{\partial J^</em>}{\partial u}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m$</td>
<td>$\frac{\partial F^<em>}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right)^</em> + \cdots$</td>
</tr>
</tbody>
</table>

**Hessian action cost**

4 equation solves per action.
Equation structure

\[ F(u, m) = 0 \quad \frac{\partial F^*}{\partial u} \lambda = \frac{\partial J^*}{\partial u} \]

\[ \frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m \quad \frac{\partial F^*}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right)^* + \cdots \]

**Hessian action cost**

2 equation solves + 2 equation solves per action.
Equation structure

\[ F(u, m) = 0 \]
\[ \frac{\partial F^*}{\partial u} \lambda = \frac{\partial J^*}{\partial u} \]
\[ \frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m \]
\[ \frac{\partial F^*}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right)^* + \cdots \]

Exploit the problem structure

Factorise the linearised equation once
Equation structure

\[ F(u, m) = 0 \]

\[ \frac{\partial F}{\partial u} \lambda = \frac{\partial J}{\partial u} \]

\[ \frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m \]

\[ \frac{\partial F}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right)^* + \cdots \]

Exploit the problem structure

Factorise the linearised equation once (if you can).
Equation structure

\[ F(u, m) = 0 \]

\[ \frac{\partial F^*}{\partial u} \lambda = \frac{\partial J^*}{\partial u} \]

\[ \frac{\partial F}{\partial u} \dot{u} = \frac{\partial F}{\partial m} \delta m \]

\[ \frac{\partial F^*}{\partial u} \dot{\lambda} = \left( \frac{\partial^2 J}{\partial u^2} \dot{u} \right)^* + \cdots \]

Hessian action cost

2 equation solves + 2 vector assemblies/substitutions per action.
Deckelnick & Hinze (2002)

\[
\min_{u,m} J(u, m) = \frac{1}{2} \|u(T) - u_d\|^2 + \frac{\alpha}{2} \|m - m_0\|^2
\]

subject to
\[
\begin{align*}
    u_t - \nu \Delta u + \nabla p &= m, \\
    \nabla \cdot u &= 0, \\
    u(\cdot, t) &= 0, \\
    u(\cdot, 0) &= u_0.
\end{align*}
\]
Deckelnick & Hinze (2002)

\[
\min_{u,m} J(u, m) = \frac{1}{2} || u(T) - u_d ||^2 + \frac{\alpha}{2} || m - m_0 ||^2
\]

subject to

\[
\begin{align*}
    u_t - \nu \Delta u + \nabla p &= m, \\
    \nabla \cdot u &= 0, \\
    u(\cdot, t) &= 0, \\
    u(\cdot, 0) &= u_0.
\end{align*}
\]

**Posterior covariance**

Compute the eigendecomposition of the Hessian at the minimiser.
Deckelnick & Hinze (2002)

Code

\[
H = \text{hessian}(J, m) \\
eig = H.\text{eigendecomposition}(n=30)
\]
**Code**

\[ H = \text{hessian}(J, m) \]
\[ \text{eig} = H.\text{eigendecomposition}(n=30) \]

<table>
<thead>
<tr>
<th></th>
<th>With caching (s)</th>
<th>Without (s)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward run</td>
<td>8.1</td>
<td>99.7</td>
<td>0.0465</td>
</tr>
<tr>
<td>Hessian action</td>
<td>8.1</td>
<td>173.9</td>
<td>0.0465</td>
</tr>
</tbody>
</table>
Deckelnick & Hinze (2002)

$m_{18}$
Deckelnick & Hinze (2002)
Deckelnick & Hinze (2002)
Deckelnick & Hinze (2002)

$m_{29}$
dolfin-adjoint

http://dolfin-adjoint.org