Imperial College London

dolfin-adjoint: automating the adjoints of models written in the Python interface to DOLFIN

David A. Ham[1,2]   Patrick E. Farrell[1]   Simon W. Funke[1,2]
Marie E. Rognes[3]

[1]Department of Earth Science and Engineering, Imperial College London
[2]Grantham Institute for Climate Change, Imperial College London
[3]Simula Research Laboratory, Lysaker, Norway

A tale of two abstractions

The fundamental abstraction of libadjoint

A model is a sequence of equations which are solved for fields.

The Python interface to DOLFIN

A model is a sequence of variational problems expressed in high-level mathematical form at run time.

Traditional algorithmic (automatic) differentiation

$$\begin{tikzcd} \boxed{\text{discrete forward equations}} \arrow[r, "\text{implement model by hand}"] & \boxed{\text{forward code}} \arrow[d, "\text{algorithmic differentiation}"'] \\ & \boxed{\text{adjoint code}} \end{tikzcd}$$

[1] Naumann, U., 2011. The Art of Differentiating Computer Programs. Software, Environments and Tools. SIAM

Traditional algorithmic (automatic) differentiation

$$\begin{tikzcd}
\boxed{\text{discrete forward equations}} \arrow[r, "\text{implement model by hand}"] & \boxed{\text{forward code}} \arrow[d, "\text{algorithmic differentiation}"'] \\
& \boxed{\text{adjoint code}}
\end{tikzcd}$$

- differentiating C or Fortran is a hard and fragile process.
- the resulting code is typically slow (3-30 times slower[1])
- implementing checkpointing in low-level code is hard
- adjoining parallelism is hard.

[1] Naumann, U., 2011. The Art of Differentiating Computer Programs. Software, Environments and Tools. SIAM

dolfin-adjoint's approach

$$
\begin{tikzcd}
\boxed{\text{discrete forward equations}} \arrow[r, "\text{FEniCS system}"] \arrow[d, "\text{dolfin-adjoint}"'] & \boxed{\text{forward code}} \\
\boxed{\text{discrete adjoint equations}} \arrow[r, "\text{FEniCS system}"] & \boxed{\text{adjoint code}}
\end{tikzcd}
$$

dolfin-adjoint's approach

$$\begin{tikzcd} \boxed{\text{discrete forward equations}} \arrow[r, "\text{FEniCS system}"] \arrow[d, "\text{dolfin-adjoint}"'] & \boxed{\text{forward code}} \\ \boxed{\text{discrete adjoint equations}} \arrow[r, "\text{FEniCS system}"] & \boxed{\text{adjoint code}} \end{tikzcd}$$

- differentiating UFL is easy (and built-in)
- resulting code is generated by the same high performance system as forward model.
- at whole equation-level, optimal checkpointing is possible.
- parallelisation comes after adjoining.

Burgers equation

```
from dolfin import *

n = 30
mesh = UnitInterval(n)
V = FunctionSpace(mesh, "CG", 2)
ic = project(Expression("sin(2*pi*x[0])"), V)
u = Function(ic)
u_next = Function(V)
v = TestFunction(V)
nu = Constant(0.0001)
timestep = Constant(1.0/n)
F = ((u_next - u)/timestep*v
     + u_next*grad(u_next)*v + nu*grad(u_next)*grad(v))*dx
bc = DirichletBC(V, 0.0, "on_boundary")
t = 0.0; end = 0.2
while (t <= end):
    solve(F == 0, u_next, bc)
    u.assign(u_next)
    t += float(timestep)
```

Burgers equation

```
from dolfin import *
from dolfin_adjoint import *
n = 30
mesh = UnitInterval(n)
V = FunctionSpace(mesh, "CG", 2)
ic = project(Expression("sin(2*pi*x[0])"), V)
u = Function(ic, name="Velocity")
u_next = Function(V, name="NextVelocity")
v = TestFunction(V)
nu = Constant(0.0001)
timestep = Constant(1.0/n)
F = ((u_next - u)/timestep*v
     + u_next*grad(u_next)*v + nu*grad(u_next)*grad(v))*dx
bc = DirichletBC(V, 0.0, "on_boundary")
t = 0.0; end = 0.2
while (t <= end):
    solve(F == 0, u_next, bc)
    u.assign(u_next)
    t += float(timestep)
    adj_inc_timestep()
```

Under the hood: overloading solve

Calls to solve (and other DOLFIN functions) are intercepted:

- Equation dependencies are extracted.
- Variables and initial conditions are registered.
- Diagonal block and RHS objects are created using the forms passed to solve.
- Values of non-linear dependencies are recorded.

Using the adjoint: FinalFunctional

```
J = FinalFunctional(0.5*inner(u, u)*dx)
ic_param = InitialConditionParameter("Velocity")
dJdic = compute_gradient(J, ic_param)
print norm(dJdic)
plot(dJdic, interactive=True)
```

Under the hood of the adjoint calculation

```
def compute_adjoint(functional, forget=True):

  for i in range(adjglobals.adjointer.equation_count)[::-1]:
      (adj_var, output) = adjglobals.adjointer.get_adjoint_solution(i, functional)

      storage = libadjoint.MemoryStorage(output)
      adjglobals.adjointer.record_variable(adj_var, storage)

      # forget is None: forget *nothing*.
      # forget is True: forget everything we can, forward and adjoint
      # forget is False: forget only unnecessary adjoint values
      if forget is None:
        pass
      elif forget:
        adjglobals.adjointer.forget_adjoint_equation(i)
      else:
        adjglobals.adjointer.forget_adjoint_values(i)

      yield (output.data, adj_var)
```

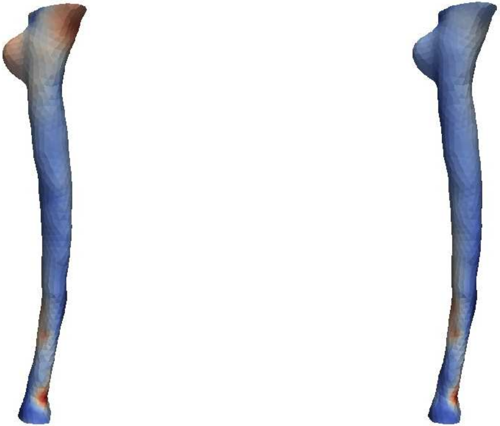
Example: visco-elastic spinal cord

The Standard Linear Solid viscoelastic model equations can be phrased as: find the Maxwell stress tensor σ_0, the elastic stress tensor σ_1, the velocity v and the vorticity γ such that

$$
\begin{aligned}
A_1^0 \frac{\partial}{\partial t}\sigma_0 + A_0^0\sigma_0 - \nabla v + \gamma &= 0, \\
A_1^1 \frac{\partial}{\partial t}\sigma_1 - \nabla v + \gamma &= 0, \\
\nabla \cdot (\sigma_0 + \sigma_1) &= 0, \\
\mathrm{skw}(\sigma_0 + \sigma_1) &= 0,
\end{aligned}
$$

for $(t; x, y, z) \in (0, T] \times \Omega$. Here, A_1^0, A_0^0, A_1^1 are fourth-order compliance tensors.

Visco-elastic spinal cord



Maxwell stress (left) and adjoint Maxwell stress (right) for visco-elastic spinal cord simulation.

Visco-elastic spinal cord performance results

	Runtime (s)	Ratio
Forward model	119.93	
Annotation	0.31	0.003
Annotation + adjoint model	124.06	1.034

Imperial College London

Demonstrably correct adjoints

δa	$\left\lvert\widehat{J}(a+\delta a)-\widehat{J}(a)\right\rvert$	order	$\left\lvert\widehat{J}(a+\delta a)-\widehat{J}(a)-\nabla\widehat{J}\cdot\delta a\right\rvert$	order
0.05	9.1012×10^{-3}		3.0337×10^{-3}	
0.025	3.7921×10^{-3}	1.2630	7.58417×10^{-4}	2.0000
0.0125	1.7064×10^{-3}	1.1520	1.8959×10^{-4}	2.0000
6.25×10^{-3}	8.0583×10^{-4}	1.0824	4.7397×10^{-5}	2.0001
3.125×10^{-3}	3.9106×10^{-4}	1.0430	1.1848×10^{-5}	2.0001

dolfin-adjoint summary

> *[T]he automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing.*

Naumann, U., 2011. The Art of Differentiating Computer Programs. Software, Environments and Tools. SIAM

dolfin-adjoint summary

> *[T]he automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing.*

Naumann, U., 2011. The Art of Differentiating Computer Programs. Software, Environments and Tools. SIAM

For a broad class of finite element models, dolfin_adjoint delivers this.

Farrell, P. E., Funke, S. W., Ham, D. A., 2012a. A new approach for developing discrete adjoint models. Submitted to ACM Transactions on Mathematical Software

Farrell, P. E., Ham, D. A., Funke, S. W., Rognes, M. E., 2012b. Automated derivation of the adjoint of high-level transient finite element programs. Submitted to SIAM Journal on Scientific Computing

`http://dolfin-adjoint.org`