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

\[ \text{discrete forward equations} \xrightarrow{\text{implement model by hand}} \text{forward code} \]

algorithmic differentiation

\[ \downarrow \]

\[ \text{adjoint code} \]

---

Traditional algorithmic (automatic) differentiation

discrete forward equations \implies \text{implement model by hand} \implies \text{forward code}

algorithmic differentiation \implies \text{adjoint code}

- differentiating C or Fortran is a hard and fragile process.
- the resulting code is typically slow (3-30 times slower\footnote{Naumann, U., 2011. The Art of Differentiating Computer Programs. Software, Environments and Tools. SIAM})
- implementing checkpointing in low-level code is hard
- adjoining parallelism is hard.

dolfin-adjoint’s approach

```
<table>
<thead>
<tr>
<th>discrete forward equations</th>
<th>FEniCS system</th>
<th>forward code</th>
</tr>
</thead>
</table>
dolfin-adjoint              |               |              |
| discrete adjoint equations| FEniCS system | adjoint code |
```

David Ham
dolfin-adjoint’s approach

- 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

```python
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

```python
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)
```

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

\[
J = \text{FinalFunctional}(0.5 \ast \text{inner}(u, u) \ast dx)
\]

ic_param = InitialConditionParameter("Velocity")

dJdic = compute_gradient(J, ic_param)

print norm(dJdic)

plot(dJdic, interactive=True)
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 $\sigma_0$, the elastic stress tensor $\sigma_1$, the velocity $\nu$ and the vorticity $\gamma$ such that

$$
A_0^1 \frac{\partial}{\partial t} \sigma_0 + A_0^0 \sigma_0 - \nabla \nu + \gamma = 0,
$$

$$
A_1^1 \frac{\partial}{\partial t} \sigma_1 - \nabla \nu + \gamma = 0,
$$

$$
\nabla \cdot (\sigma_0 + \sigma_1) = 0,
$$

$$
\text{skw}(\sigma_0 + \sigma_1) = 0,
$$

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

<table>
<thead>
<tr>
<th></th>
<th>Runtime (s)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward model</td>
<td>119.93</td>
<td></td>
</tr>
<tr>
<td>Annotation</td>
<td>0.31</td>
<td>0.003</td>
</tr>
<tr>
<td>Annotation + adjoint model</td>
<td>124.06</td>
<td>1.034</td>
</tr>
</tbody>
</table>
Demonstrably correct adjoints

| $\delta a$ | $|\hat{J}(a + \delta a) - \hat{J}(a)|$ | order | $|\hat{J}(a + \delta a) - \hat{J}(a) - \nabla \hat{J} \cdot \delta a|$ | order |
|---|---|---|---|---|
| 0.05 | $9.1012 \times 10^{-3}$ | | $3.0337 \times 10^{-3}$ | |
| 0.025 | $3.7921 \times 10^{-3}$ | 1.2630 | $7.58417 \times 10^{-4}$ | 2.0000 |
| 0.0125 | $1.7064 \times 10^{-3}$ | 1.1520 | $1.8959 \times 10^{-4}$ | 2.0000 |
| $6.25 \times 10^{-3}$ | $8.0583 \times 10^{-4}$ | 1.0824 | $4.7397 \times 10^{-5}$ | 2.0001 |
| $3.125 \times 10^{-3}$ | $3.9106 \times 10^{-4}$ | 1.0430 | $1.1848 \times 10^{-5}$ | 2.0001 |
dolfin-adjoint summary

-The 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.

dolfin-adjoint summary

[The 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.]


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


http://dolfin-adjoint.org