Imperial College London

Fully automatic adjoints: a robust and efficient mechanism for generating adjoint dynamical cores

David A. Ham[1,3] Patrick E. Farrell[2] Simon W. Funke[2,3]
Marie E. Rognes[4]

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

Automated adjoints for finite element models

Simulations are functions which yield system state given control parameters (eg initial and boundary conditions).

$$u = f(m)$$

or equivalently

$$F(m, u) = 0$$

where u is the system state and m are the controls.

Adjoint problems

Usually we aren't interested in the whole output state but in some functional of it $J(u, m)$. Examples:

- the power output of a simulated turbine,
- the misfit from a set of observations,
- the arrival time of a tsunami.

Adjoint problems

In many of these cases, we'd like either:

$$\frac{\mathrm{d}u}{\mathrm{d}m}$$

or

$$\frac{\mathrm{d}J(u,m)}{\mathrm{d}m}$$

With these we can:

- conduct sensitivity analysis
- conduct generalised stability analysis
- employ efficient optimisation routines
- assimilate data

Adjoint problems

We can "brute force" $\mathrm{d}u/\mathrm{d}m$ using finite differences:

$$\frac{\mathrm{d}u}{\mathrm{d}m_i} \approx \frac{f(m + h\delta m_i) - f(m)}{h}$$

for each m_i.
. . . or we can solve the adjoint equations:

$$F_u^*(m, u)\lambda = J_u^*(u, m)$$

and then calculating:

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\lambda^* F_m + J_m$$

General form of the transient adjoint problem

Forward system:

$$\begin{pmatrix} I & & & & \\ T(u_0) & M & & & \\ & T(u_1) & M & & \\ & & \ddots & \ddots & \\ & & & T(u_{N-1}) & M \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g \\ \Delta tf_0 \\ \Delta tf_1 \\ \vdots \\ \Delta tf_{N-1} \end{pmatrix}$$

General form of the transient adjoint problem

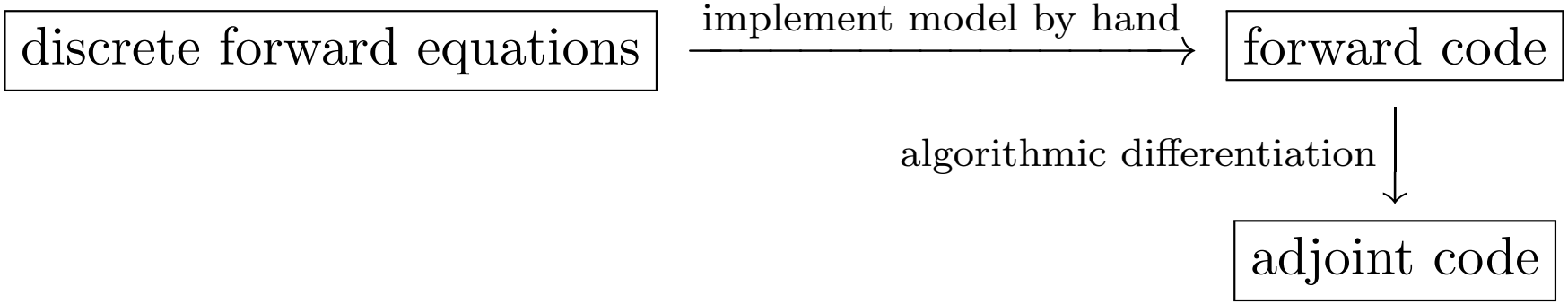
Forward system:

$$\begin{pmatrix} I & & & & \\ T(u_0) & M & & & \\ & T(u_1) & M & & \\ & & \ddots & \ddots & \\ & & & T(u_{N-1}) & M \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g \\ \Delta tf_0 \\ \Delta tf_1 \\ \vdots \\ \Delta tf_{N-1} \end{pmatrix}$$

Adjoint system:

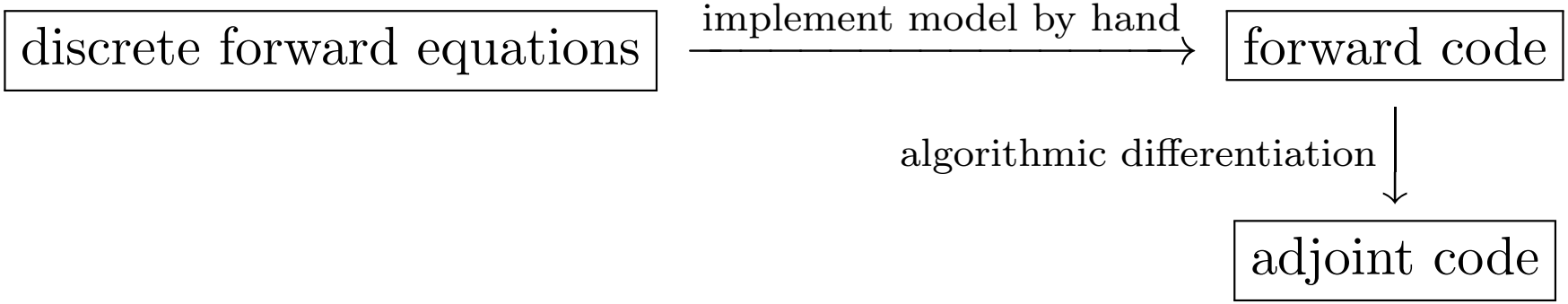
$$\begin{pmatrix} I^* & \left(T(u_0) + \frac{\partial T(u_0)}{\partial u_0} u_0\right)^* & & & \\ & M^* & \left(T(u_1) + \frac{\partial T(u_1)}{\partial u_1} u_1\right)^* & & \\ & & \ddots & \ddots & \\ & & & \ddots & \ddots \\ & & & & M^* \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ \vdots \\ z_N \end{pmatrix} = \frac{\partial J}{\partial u}.$$

Traditional algorithmic automatic differentiation



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

Traditional algorithmic automatic differentiation



- 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

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

Unified Form Language

Consider Poisson's equation in weak form:

$$\int_\Omega \nabla\phi \cdot \nabla\psi \mathrm{d}x = \int_\Omega \phi f \mathrm{d}x$$

```
phi=TestFunction(Psi)
psi=TrialFunction(Psi)
lhs=dot(grad(phi),grad(psi))*dx
rhs=phi*f*dx
Psi=solve(lhs,rhs)
```

UFL was developed by Marten Alnæs and Anders Logg for the FEniCS project.

Slightly less trivial example: C-grid linear shallow water equations

```
V = FunctionSpace(mesh, 'Raviart-Thomas', 1)
H = FunctionSpace(mesh, 'DG', 0)
W = V*H
(v, q) = TestFunctions(W)
(u, h) = TrialFunctions(W)
M_u = inner(v,u)*dx
M_h = q*h*dx
Ct = -inner(avg(u),jump(q,n))*dS
C = c**2*inner(avg(v),jump(h,n))*dS
F = f*inner(v,as_vector([-u[1],u[0]]))*dx
A = assemble(M_u+M_h+0.5*dt*(C-Ct+F))
A_r = M_u+M_h-0.5*dt*(C-Ct+F)
```

Slightly less trivial example: C-grid linear shallow water equations

Maths as code. The divergence and pressure gradient:

$$\int_\Omega q \nabla \cdot \mathbf{u} \, \mathrm{d}V = -\int_{\Gamma E} \mathbf{u} \cdot \mathbf{n} (q^+ - q^-) \, \mathrm{d}S$$

$$c^2 \int_\Omega \mathbf{v} \cdot \nabla h \, \mathrm{d}V = c^2 \int_{\Gamma E} (h^+ - h^-) \mathbf{n} \cdot \mathbf{v} \, \mathrm{d}S$$

become:

```
Ct = -inner(avg(u),jump(q,n))*dS
C = c**2*inner(avg(v),jump(h,n))*dS
```

General form of the transient adjoint problem

Forward system:

$$\begin{pmatrix} I & & & & \\ T(u_0) & M & & & \\ & T(u_1) & M & & \\ & & \ddots & \ddots & \\ & & & T(u_{N-1}) & M \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g \\ \Delta tf_0 \\ \Delta tf_1 \\ \vdots \\ \Delta tf_{N-1} \end{pmatrix}$$

General form of the transient adjoint problem

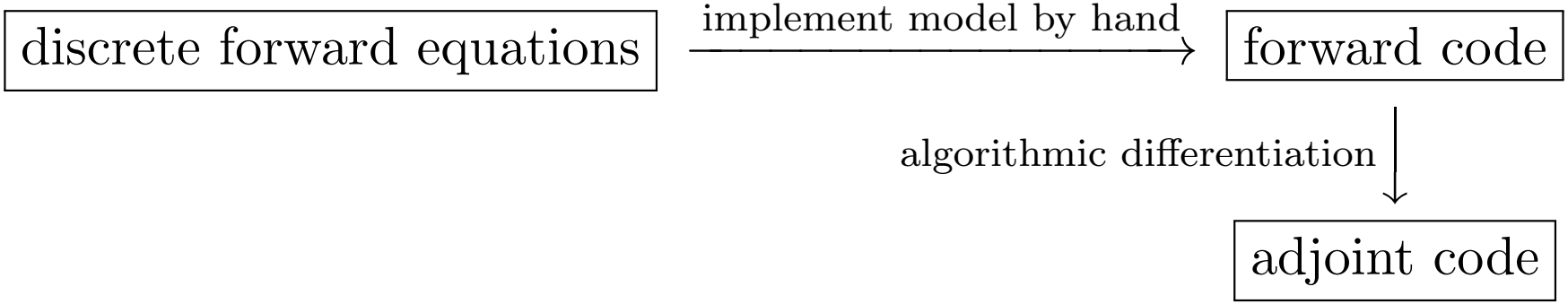
Forward system:

$$\begin{pmatrix} I & & & & \\ T(u_0) & M & & & \\ & T(u_1) & M & & \\ & & \ddots & \ddots & \\ & & & T(u_{N-1}) & M \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_N \end{pmatrix} = \begin{pmatrix} g \\ \Delta tf_0 \\ \Delta tf_1 \\ \vdots \\ \Delta tf_{N-1} \end{pmatrix}$$

Adjoint system:

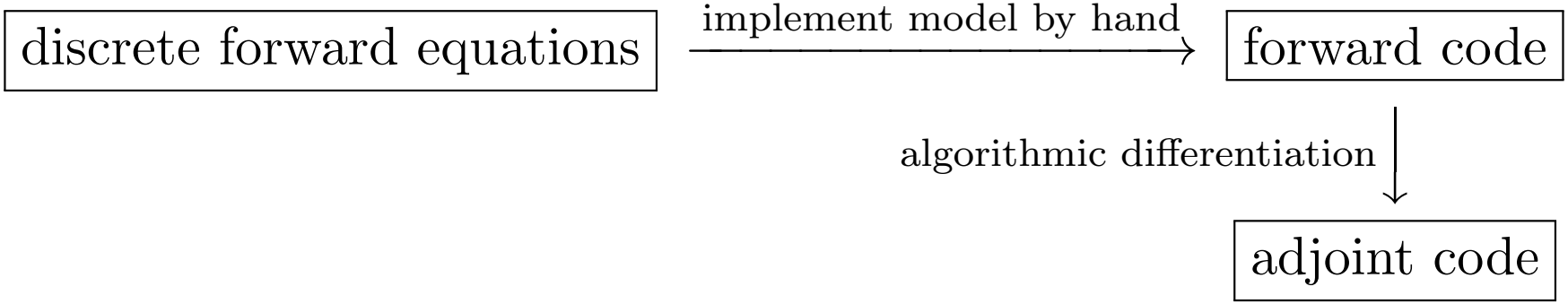
$$\begin{pmatrix} I^* & \left(T(u_0) + \frac{\partial T(u_0)}{\partial u_0} u_0\right)^* & & & \\ & M^* & \left(T(u_1) + \frac{\partial T(u_1)}{\partial u_1} u_1\right)^* & & \\ & & \ddots & \ddots & \\ & & & \ddots & \ddots \\ & & & & M^* \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ \vdots \\ z_N \end{pmatrix} = \frac{\partial J}{\partial u}.$$

Traditional algorithmic automatic differentiation



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

Traditional algorithmic automatic differentiation



- differentiating C or Fortran is a hard and fragile process.
- the resulting code is typically slow (3-30 times slower[2])
- implementing checkpointing in low-level code is hard

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

A tale of two abstractions

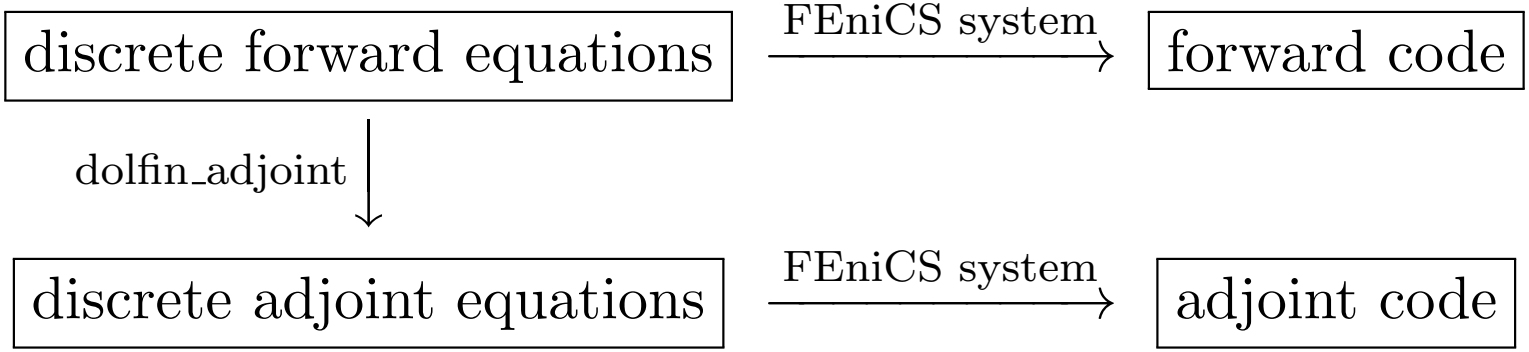
The fundamental abstraction of libadjoint

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

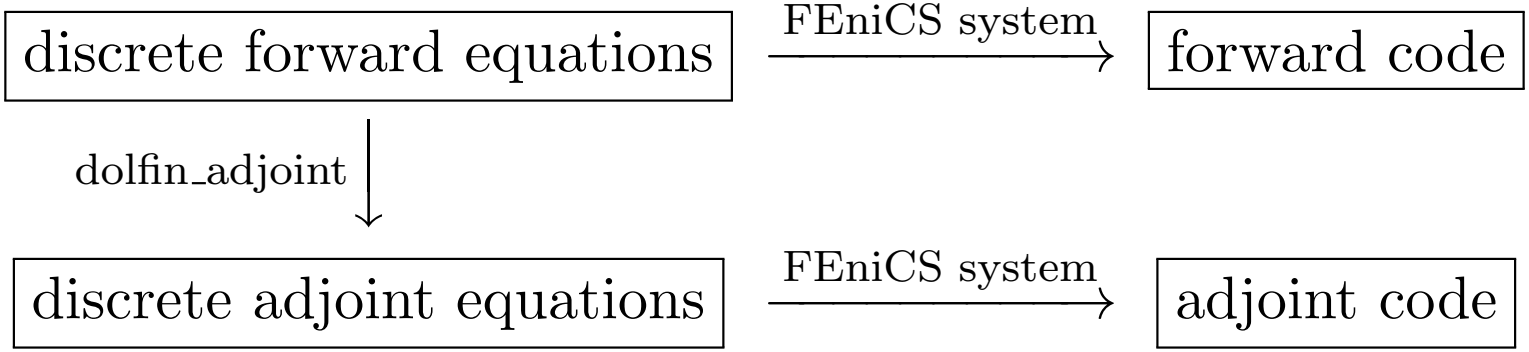
The Python interface to DOLFIN

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

dolfin_adjoint's approach



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.

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()
```

Using the adjoint: Functionals

```
J = Functional(0.5*inner(u, u)*dx*dt[FINISH_TIME])
dJdic = compute_gradient(J, ScalarParameter("Velocity"))
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)
```


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

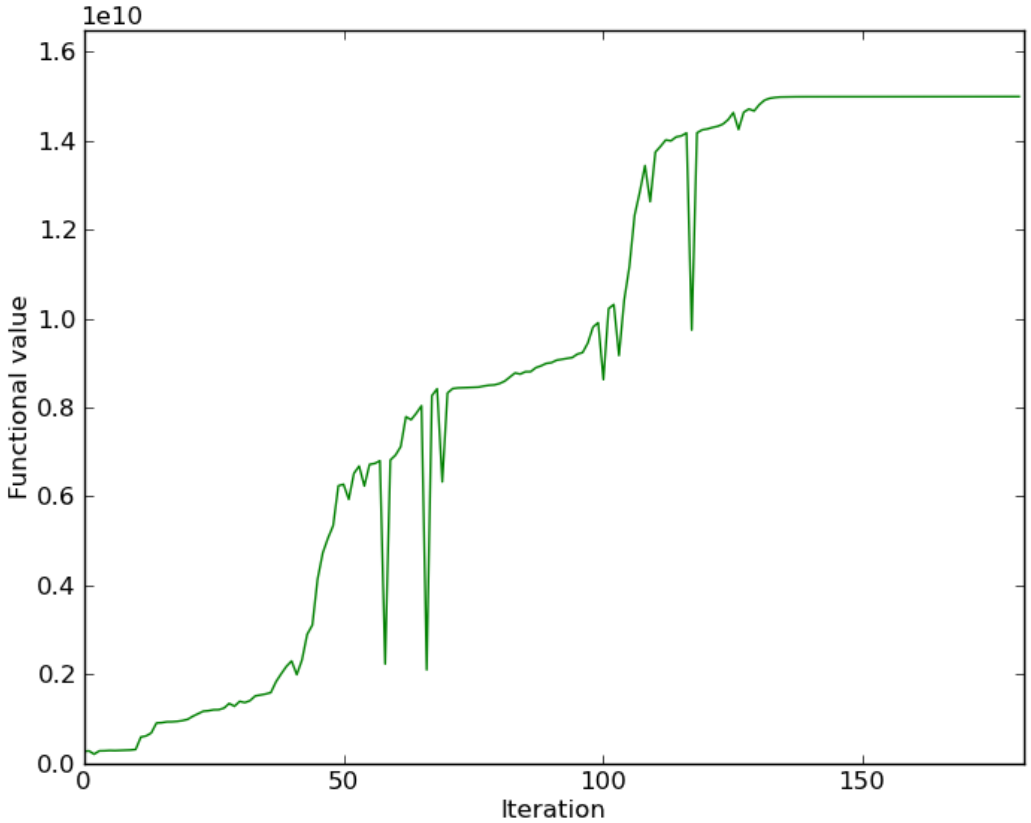
Demonstrably correct adjoints

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

Marine power optimisation example

- Shallow water equations.
- Simple parameterised turbines (drag $= c|u|^3$).
- Optimisation outsourced to an off-the-shelf Python library.

Marine power optimisation example



Side note: what if my model isn't finite element?

A similar process, using a bit of AD, is almost certainly possible for models written in a kernel/access descriptor manner:

```
pressure_gradient_kernel = op2.Kernel("""
#include <math.h>
void pressure_gradient(double *dp, double *x[2], double **p){
  *dp = (p[1][0] - p[0][0])/sqrt(pow(x[1][1]-x[0][1], 2)
                               + pow(x[1][0]-x[0][0], 2))
}""", "pressure_gradient")

op2.parloop(pressure_gradient_kernel, facets,
            pressure_gradient(op2.IdentityMap, op2.WRITE),
            cell_coordinate(facets2cells, op2.READ),
            pressure(facets2cells, op2.READ)
            )
```

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., 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`