FEniCS Course

Lecture 13: Introduction to dolfin-adjoint

Contributors
Simon Funke
Patrick Farrell
Marie E. Rognes

dolfin-adjoint
automatic adjoint models for FEniCS

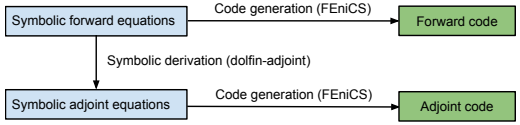
P. E. Farrell, S. W. Funke, D. A. Ham, M. E. Rognes
Simon@simula.no

The dolfin-adjoint project automatically derives and solves adjoint and tangent linear equations from high-level mathematical specifications of finite element discretizations of partial differential equations.

About dolfin-adjoint

How it works

Adjoint and tangent linear models form the basis of many numerical techniques such as sensitivity analysis, optimization, and stability analysis. However, the derivation and implementation of adjoint models for nonlinear or time-dependent models are notoriously challenging: the manual approach is time-consuming and error-prone and traditional automatic differentiation tools lack robustness and performance.
dolfin-adjoint solves this problem by automatically analyzing the high-level mathematical structure inherent in finite element methods. It raises the traditional abstraction of algorithmic differentiation from the level of individual floating point operations to that of whole systems of differential equations. This approach delivers a number of advantages over the previous state-of-the-art: robust hands-off automation of adjoint model derivation; computational efficiency approaching the theoretical optimum; and native parallel support inherited from the forward model.



◁ **Figure**: By adding a few lines of code to an existing FEniCS model, dolfin-adjoint computes tangent linear and adjoint solutions, gradients and Hessian actions of arbitrary user-specified functionals, and uses these derivatives in combination with sophisticated optimization algorithms or to conduct stability analyses

The implementation of dolfin-adjoint is based on the finite-element framework FEniCS. When the user runs a FEniCS model, dolfin-adjoint records the dependencies and structure of the forward equations. The resulting execution graph stores a mathematical representation of the forward equations. By reasoning about this graph, dolfin-adjoint can linearize the equations to derive a symbolic representation of the discrete tangent linear equations, and reverse the propagation of information to derive the corresponding adjoint equations. By invoking the FEniCS automatic code generator on these equations, dolfin-adjoint obtains solutions of the tangent linear and adjoint models, and can use these to compute consistent first and second order functional derivatives. dolfin-adjoint also has preliminary support for the Firedrake project.

Performance

dolfin-adjoint runs naturally in parallel, and inherits the scalability and code optimizations of FEniCS. To verify this, we benchmarked the sensitivity analysis and generalized stability application examples.

Sensitivity analysis example

CPUs	1	2	4	Optimal
Forward runtime (s)	40.3	19.6	13.2	
Adjoint runtime (s)	39.1	19.3	12.5	
Adjoint/Forward ratio	0.97	0.99	0.95	1.00

Generalized stability example

CPUs	1	2	Optimal
Forward runtime (s)	92.4	55.0	
Adjoint runtime (s)	41.4	25.9	
Adjoint/Forward ratio	0.45	0.47	0.5

Tables: The sensitivity analysis example is linear, while the generalized stability analysis example is nonlinear and converges on average in 2 Newton-iteration per timestep. Hence the adjoint model is expected to be twice as fast as the forward model.

Checkpointing

The adjoint equations depend on the forward solutions. However, storing the entire forward trajectory is infeasible for large, time-dependent simulations. In this case, dolfin-adjoint can employ a binomial checkpointing strategy via the revolve library. When activated, dolfin-adjoint automatically saves state checkpoints and uses them to recompute missing forward states to trade off memory requirements and computational effort. This allows for solving adjoint equations even for large-scale simulations. For instance, 390 checkpoints allow simulations with 10^7 time-steps at a cost of a 3× slow-down.

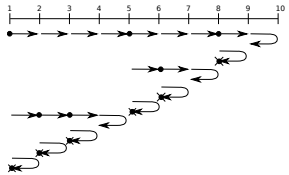


Figure: Visualisation of the optimal checkpointing strategy with 10 time levels and 3 checkpoints

Application examples

Sensitivity analysis

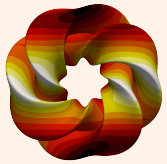
Consider the time dependent heat equation

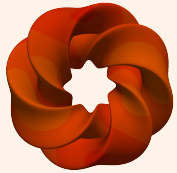
$$\frac{\partial u}{\partial t} - \nu\nabla^2 u = 0 \quad \text{in } \Omega \times (0, T),$$
$$u = g \quad \text{for } \Omega \times \{0\}.$$

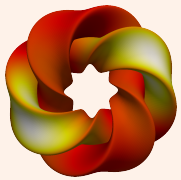
Here Ω is the Gray's Klein bottle, a closed 2D manifold embedded in 3D, T is the final time, u is the unknown temperature, ν is the thermal diffusivity, and g is the initial temperature.
The goal is to compute the sensitivity of the norm of temperature at the final time

$$J(u) = \int_\Omega u(t = T)^2$$

with respect to the initial temperature, that is $\mathrm{d}J/\mathrm{d}g$.


Initial temperature


Final temperature


Sensitivity

```
from dolfin import *
from dolfin_adjoint import *

# Solve the forward system
F = u*v*dx - u_old*v*dx +
    dt*nu*inner(grad(v),grad(u))*dx
while t <= T:
    t += dt
    solve(F == 0, u)

# Apply dolfin-adjoint
m = Control(g)
J = u**2*dx*dt[T]
dJdm = compute_gradient(J, m)
H = hessian(J, m)
```

△
Code: Implementation excerpt (the code including the complete forward model has 37 lines)

PDE-constrained optimization

This topology optimization example minimizes the compliance

$$\int_\Omega fT + \alpha \int_\Omega \nabla a \cdot \nabla a,$$

subject to the Poisson equation with mixed Dirichlet–Neumann conditions

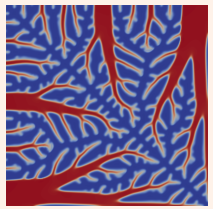
$$-\mathrm{div}(k(a)\nabla T) = f \quad \text{in } \Omega,$$
$$T = 0 \quad \text{on } \partial\Omega_D,$$
$$k(a)\nabla T = 0 \quad \text{on } \partial\Omega_N,$$

and additional control constraints

$$\int_\Omega a \le V \text{ and } 0 \le a(x) \le 1 \qquad \forall x \in \Omega.$$

Here Ω is the unit square, T is the temperature, a is the control ($a(x) = 1$ means material, $a(x) = 0$ means no material), f is a source term, $k(a)$ is the Solid Isotropic Material with Penalisation parameterization, α is a regularization term, and V is the volume bound on the control. Physically, the problem is to find the material distribution a that minimizes the integral of the temperature for a limited amount of conducting material.

```
from dolfin import *
from dolfin_adjoint import *
# ...
J = f*T*dx + alpha*inner(grad(a),grad(a))*dx
m = Control(a)
rf = ReducedFunctional(J, m)
minimize(rf, method="SLSQP", bounds=...)
```



△
Code: Implementation excerpt (the full code uses the IPOPT optimization package and has 56 lines)
◁ **Figure**: Optimal material distribution a for a unit square domain and $f = 10^{-2}$

Generalized stability analysis

This example performs a generalized stability analysis to find the perturbations to an initial condition that grow the most over some finite time. The governing equations are the two-dimensional vorticity-streamfunction formulation of the time-dependent Navier–Stokes equations, coupled to two advection equations for temperature and salinity:

Adjoints are key ingredients for sensitivity analysis, PDE-constrained optimization, ...

So far we have focused on solving forward PDEs.

But we want to do (and can do) more than that!

Maybe we are interested in ...

- the sensitivity with respect to certain parameters
  - initial conditions,
  - forcing terms,
  - unknown coefficients.
- PDE-constrained optimization
  - data assimilation
  - optimal control
- goal-oriented error control

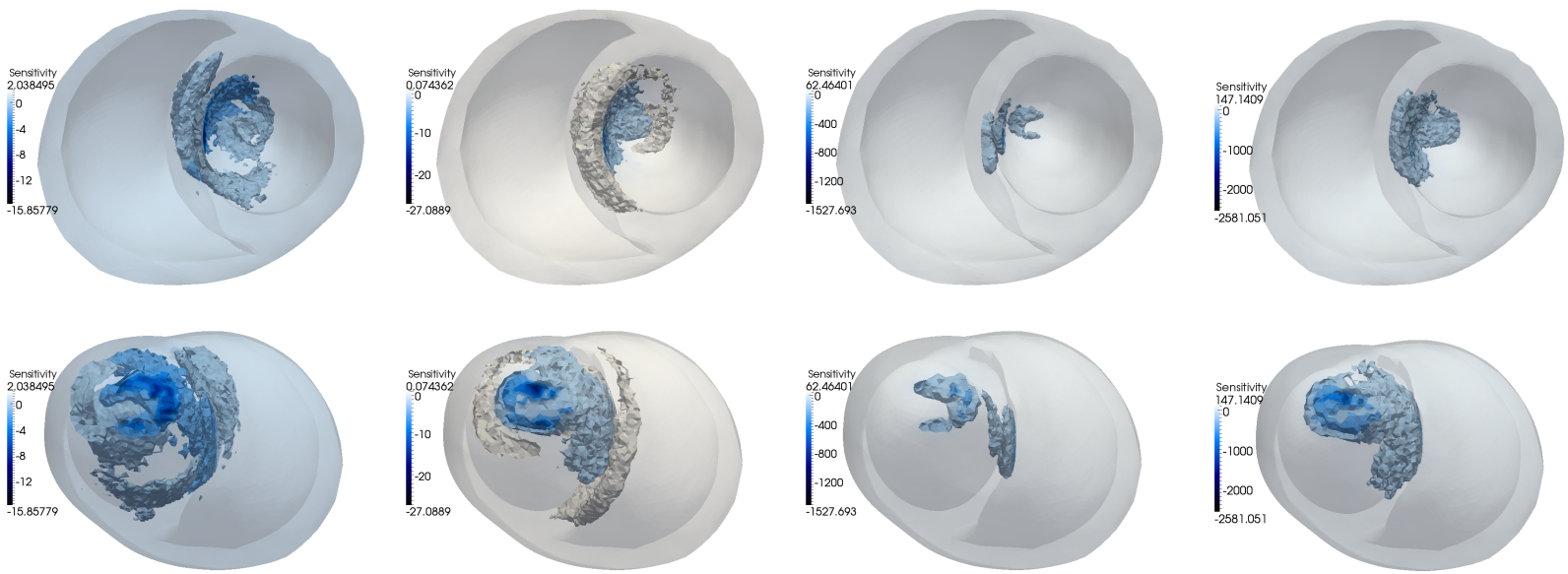
For this we want to compute functional derivatives and adjoints provide an efficient way of doing so.

What is the sensitivity of the abnormal wave propagation to the local tissue conductivities?

The wave propagation abnormality at a given time T:

$$J(v, s, u) = \|v(T) - v_{\text{obs}}(T)\|^2, \qquad \frac{\partial J}{\partial g_{\text{e|i|l|t}}} = ?$$

```
v_d = Function(V, "healthy_obs_200.xml.gz")
J = Functional(inner(v - v_d, v - v_d)*dx*dt[T])
dJdg_s = compute_gradient(J, gs)
```



The Hello World of functional derivatives

Consider the Poisson's equation

$$\begin{aligned} -\nu\Delta u &= m \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

together with the *objective functional*

$$J(u) = \frac{1}{2}\int_\Omega \|u - u_d\|^2 \,\mathrm{d}x,$$

where u_d is a known function.

Goal

Compute the sensitivity of J with respect to the *parameter* m: $\mathrm{d}J/\mathrm{d}m$.

Computing functional derivatives (Part 1/3)

Given

- Parameter m,
- PDE $F(u, m) = 0$ with solution u.
- Objective functional $J(u, m) \to \mathbb{R}$,

Goal

Compute $\mathrm{d}J/\mathrm{d}m$.

Reduced functional

Consider u as an implicit function of m by solving the PDE.
With that we define the *reduced functional* R:

$$R(m) = J(u(m), m)$$

Computing functional derivatives (Part 1/3)

Given

- Parameter m,
- PDE $F(u, m) = 0$ with solution u.
- Objective functional $J(u, m) \to \mathbb{R}$,

Goal

Compute $\mathrm{d}J/\mathrm{d}m$.

Reduced functional

Consider u as an implicit function of m by solving the PDE.
With that we define the *reduced functional* R:

$$R(m) = J(u(m), m)$$

Computing functional derivatives (Part 1/3)

Given

- Parameter m,
- PDE $F(u, m) = 0$ with solution u.
- Objective functional $J(u, m) \to \mathbb{R}$,

Goal

Compute $\mathrm{d}J/\mathrm{d}m$.

Reduced functional

Consider u as an implicit function of m by solving the PDE.
With that we define the *reduced functional* R:

$$R(m) = J(u(m), m)$$

Computing functional derivatives (Part 2/3)

Reduced functional:

$$R(m) \equiv J(u(m), m).$$

Taking the derivative of with respect to m yields:

$$\frac{\mathrm{d}R}{\mathrm{d}m} = \frac{\mathrm{d}J}{\mathrm{d}m} = \frac{\partial J}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial J}{\partial m}.$$

Computing $\frac{\partial J}{\partial u}$ and $\frac{\partial J}{\partial m}$ is straight-forward, but how handle $\frac{\mathrm{d}u}{\mathrm{d}m}$?

Computing functional derivatives (Part 2/3)

Reduced functional:

$$R(m) \equiv J(u(m), m).$$

Taking the derivative of with respect to m yields:

$$\frac{\mathrm{d}R}{\mathrm{d}m} = \frac{\mathrm{d}J}{\mathrm{d}m} = \frac{\partial J}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial J}{\partial m}.$$

Computing $\frac{\partial J}{\partial u}$ and $\frac{\partial J}{\partial m}$ is straight-forward, but how handle $\frac{\mathrm{d}u}{\mathrm{d}m}$?

Computing functional derivatives (Part 2/3)

Reduced functional:

$$R(m) \equiv J(u(m), m).$$

Taking the derivative of with respect to m yields:

$$\frac{\mathrm{d}R}{\mathrm{d}m} = \frac{\mathrm{d}J}{\mathrm{d}m} = \frac{\partial J}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial J}{\partial m}.$$

Computing $\frac{\partial J}{\partial u}$ and $\frac{\partial J}{\partial m}$ is straight-forward, but how handle $\frac{\mathrm{d}u}{\mathrm{d}m}$?

Computing functional derivatives (Part 3/3)

Taking the derivative of $F(u, m) = 0$ with respect to m yields:

$$\frac{\mathrm{d}F}{\mathrm{d}m} = \frac{\partial F}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial F}{\partial m} = 0$$

Hence:

$$\frac{\mathrm{d}u}{\mathrm{d}m} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}$$

Final formula for functional derivative

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\overbrace{\frac{\partial J}{\partial u}\underbrace{\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}}_{\text{tangent linear PDE}}}^{\text{adjoint PDE}} + \frac{\partial J}{\partial m},$$

Computing functional derivatives (Part 3/3)

Taking the derivative of $F(u, m) = 0$ with respect to m yields:

$$\frac{\mathrm{d}F}{\mathrm{d}m} = \frac{\partial F}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial F}{\partial m} = 0$$

Hence:

$$\frac{\mathrm{d}u}{\mathrm{d}m} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}$$

Final formula for functional derivative

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\overbrace{\frac{\partial J}{\partial u}\underbrace{\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}}_{\text{tangent linear PDE}}}^{\text{adjoint PDE}} + \frac{\partial J}{\partial m},$$

Computing functional derivatives (Part 3/3)

Taking the derivative of $F(u, m) = 0$ with respect to m yields:

$$\frac{\mathrm{d}F}{\mathrm{d}m} = \frac{\partial F}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}m} + \frac{\partial F}{\partial m} = 0$$

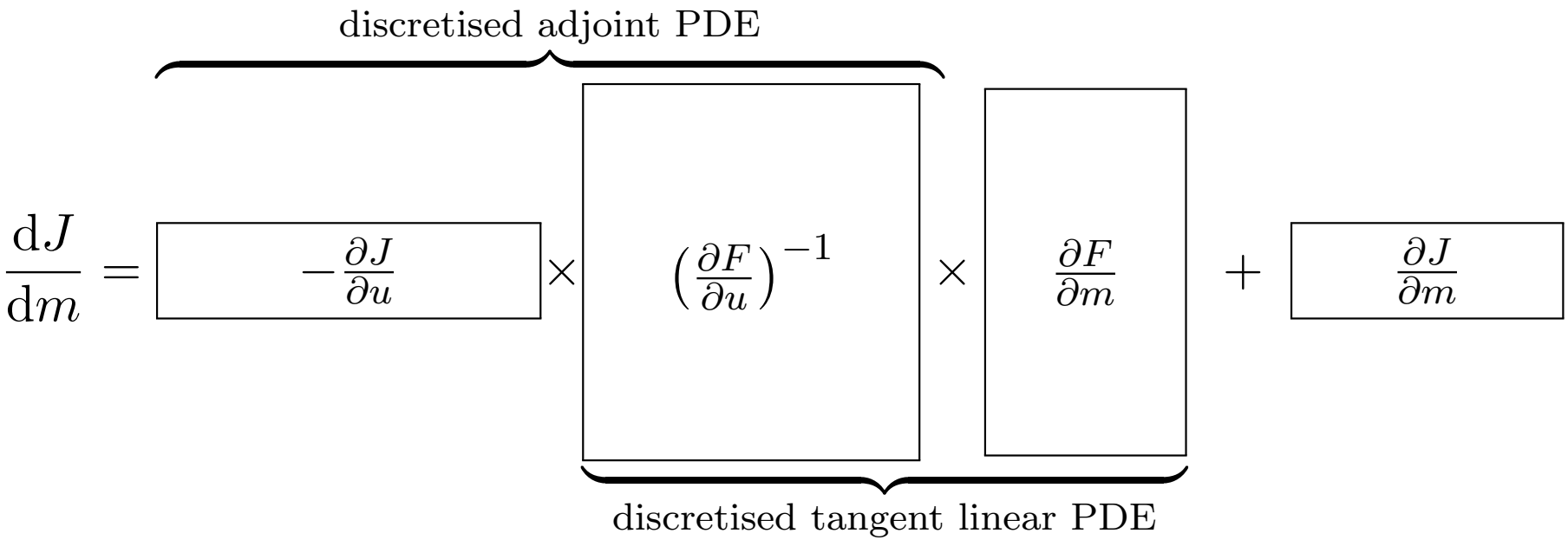
Hence:

$$\frac{\mathrm{d}u}{\mathrm{d}m} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}$$

Final formula for functional derivative

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -\overbrace{\frac{\partial J}{\partial u}\underbrace{\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m}}_{\text{tangent linear PDE}}}^{\text{adjoint PDE}} + \frac{\partial J}{\partial m},$$

Dimensions of a finite dimensional example



The tangent linear solution is a matrix of dimension $|u| \times |m|$ and requires the solution of m linear systems.

The adjoint solution is a vector of dimension $|u|$ and requires the solution of one linear system.

Adjoint approach

1. Solve the adjoint equation for λ

$$\frac{\partial F}{\partial u}^* \lambda = -\frac{\partial J^*}{\partial u}.$$

2. Compute

$$\frac{\mathrm{d}J}{\mathrm{d}m} = \lambda^* \frac{\partial F}{\partial m} + \frac{\partial J}{\partial m}.$$

The computational expensive part is (1). It requires solving the (linear) adjoint PDE, and its cost is independent of the choice of parameter m.

What is dolfin-adjoint?

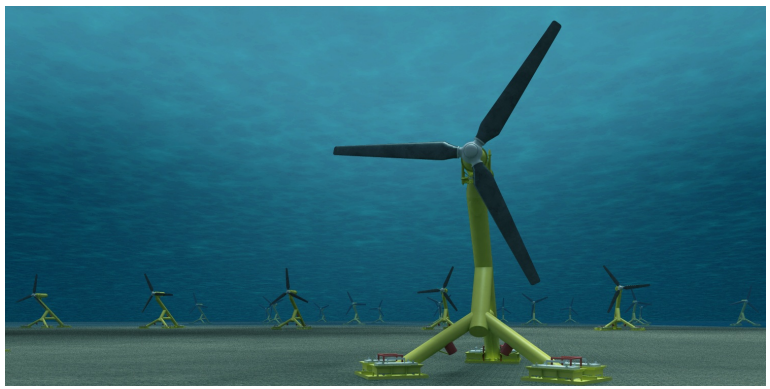
Dolfin-adjoint is an extension of FEniCS for: solving adjoint and tangent linear equations; generalised stability analysis; PDE-constrained optimisation.

Main features

- Automated derivation of first and second order adjoint and tangent linear models.
- Discretely consistent derivatives.
- Parallel support and near theoretically optimal performance.
- Interface to optimisation algorithms for PDE-constrained optimisation.
- Documentation and examples on `www.dolfin-adjoint.org`.

What has dolfin-adjoint been used for?

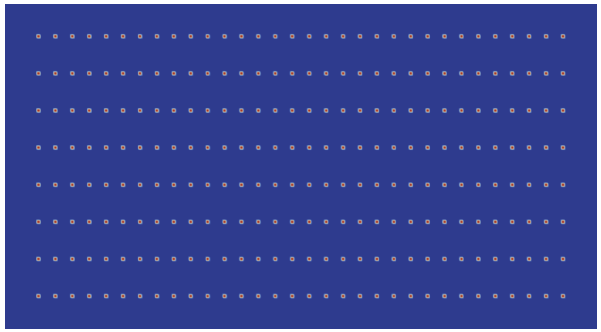
Layout optimisation of tidal turbines

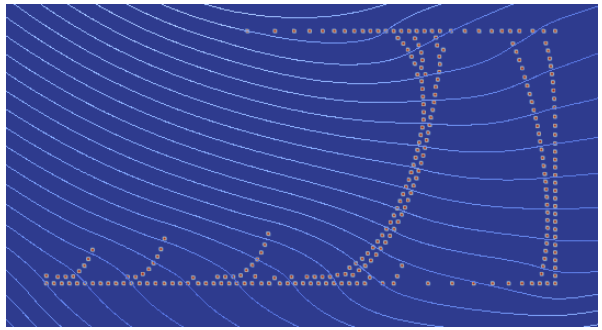


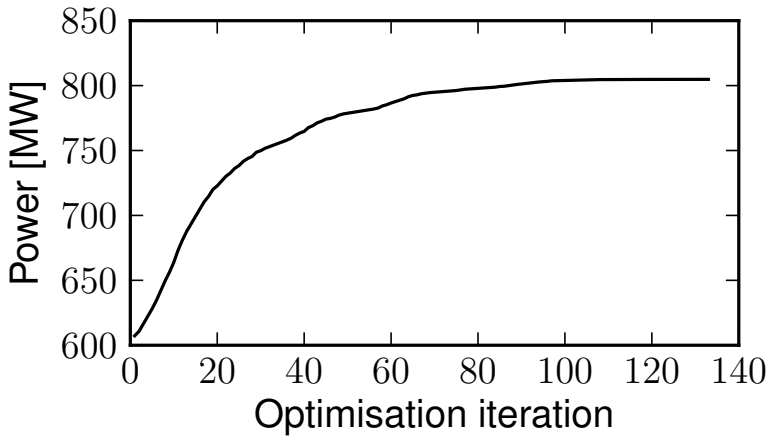
- Up to 400 tidal turbines in one farm.
- What are the optimal locations to maximise power production?

What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines







What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines

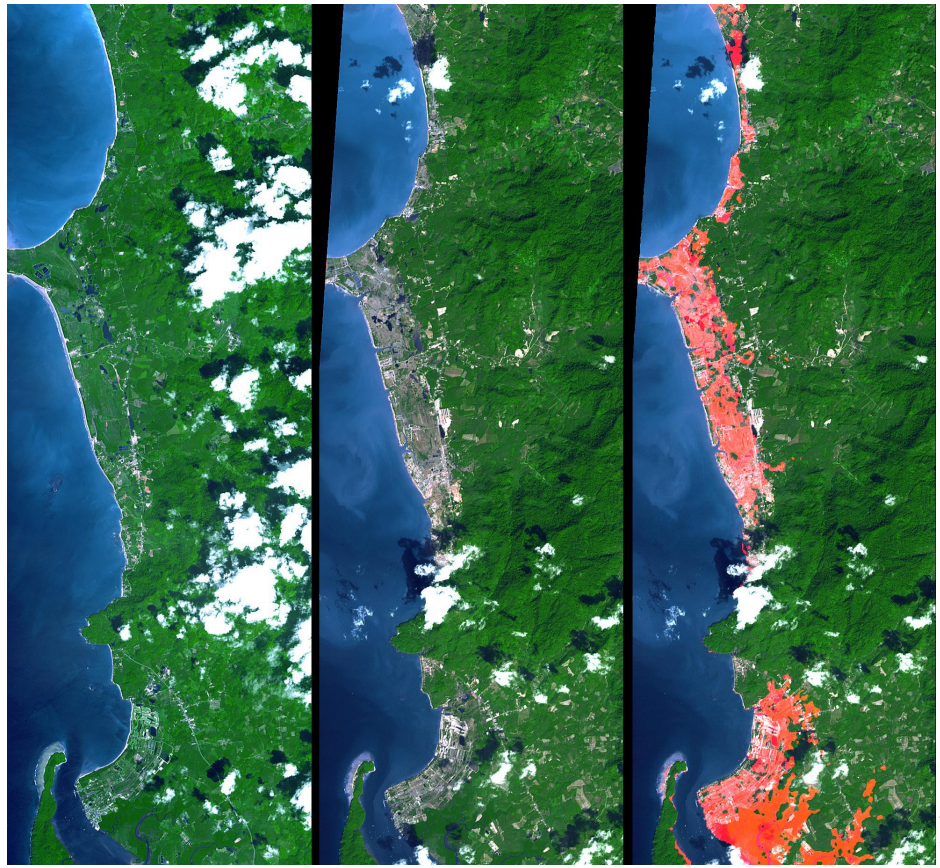
```
from dolfin import *
from dolfin_adjoint import *

# FEniCS model
# ...

J = Functional(turbines*inner(u, u)**(3/2)*dx*dt)
m = Control(turbine_positions)
R = ReducedFunctional(J, m)
maximize(R)
```

What has dolfin-adjoint been used for?

Reconstruction of a tsunami wave

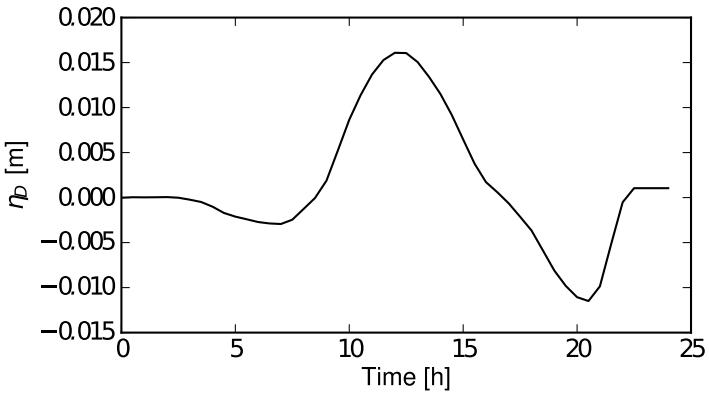
 [1]

Is it possible to reconstruct a tsunami wave from images like this?

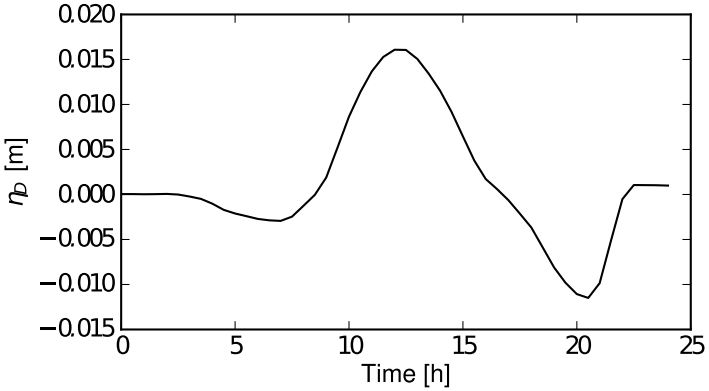
[1] Image: ASTER/NASA PIA06671

What has dolfin-adjoint been used for?

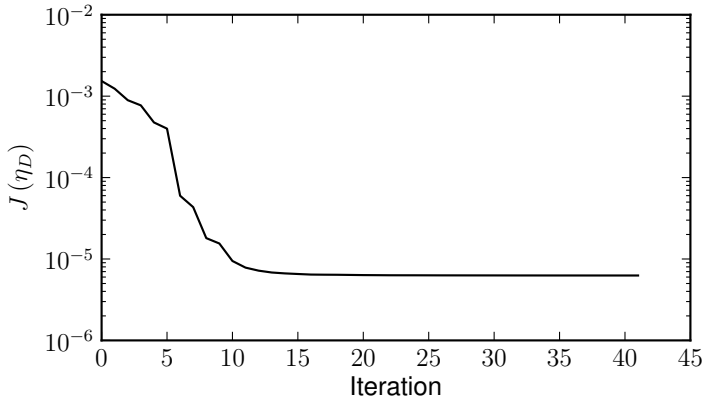
Reconstruction of a tsunami wave



Correct tsunami wave



Reconstructed tsunami wave



Reconstruction of a tsunami wave

```
from fenics import *
from dolfin_adjoint import *

# FEniCS model
# ...

J = Functional(observation_error**2*dx*dt)
m = Control(input_wave)
R = ReducedFunctional(J, m)
minimize(R)
```

Other applications

Dolfin-adjoint has been applied to lots of other cases, and **works for many PDEs**:

Some PDEs we have adjoined

- Burgers
- Navier-Stokes
- Stokes + mantle rheology
- Stokes + ice rheology
- Saint Venant + wetting/drying
- Cahn-Hilliard
- Gray-Scott
- Shallow ice
- Blatter-Pattyn
- Quasi-geostrophic
- Viscoelasticity
- Gross-Pitaevskii
- Yamabe
- Image registration
- Bidomain
- ...

Consider again our first example

Compute the sensitivity $\partial J/\partial m$ of

$$J(u) = \int_\Omega \|u - u_d\|^2 \, \mathrm{d}x$$

with known u_d and the Poisson equation:

$$\begin{aligned} -\nu \Delta u &= m \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned}$$

with respect to m.

Poisson solver in FEniCS

An implementation of the Poisson's equation might look like this:

```
from fenics import *

# Define mesh and finite element space
mesh = UnitSquareMesh(50, 50)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define basis functions and parameters
u = TrialFunction(V)
v = TestFunction(V)
m = interpolate(Constant(1.0), V)
nu = Constant(1.0)

# Define variational problem
a = nu*inner(grad(u), grad(v))*dx
L = m*v*dx
bc = DirichletBC(V, 0.0, "on_boundary")

# Solve variational problem
u = Function(V)
solve(a == L, u, bc)
plot(u, title="u")
```

Dolfin-adjoint records all relevant steps of your FEniCS code

The first change necessary to adjoin this code is to import the `dolfin-adjoint` module *after* importing `DOLFIN`:

```
from fenics import *
from dolfin_adjoint import *
```

With this, dolfin-adjoint will record each step of the model, building an *annotation*. The annotation is used to symbolically manipulate the recorded equations to derive the tangent linear and adjoint models.

In this particular example, the `solve` function method will be recorded.

Dolfin-adjoint extends the FEniCS syntax for defining objective functionals

Next, we implement the objective functional, the square L^2-norm of $u - u_d$:

$$J(u) = \int_\Omega \|u - u_d\|^2 \, \mathrm{d}x$$

or in code

```
j = inner(u - u_d, u - u_d)*dx
J = Functional(j)
```

Specify which parameter you want to differentiate with respect to using Controls

Next we need to decide which parameter we are interested in. Here, we would like to investigate the sensitivity with respect to the source term m.

We inform dolfin-adjoint of this:

```
mc = Control(m)
```

One line of code for efficiently computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, mc, project=True)
```

Dolfin-adjoint derives and solves the adjoint equations for us and returns the gradient.

Note

If you call **compute_gradient** more than once, you need to pass *forget=False* as a parameter. Otherwise you get an error: *Need a value for u_1:0:0:Forward, but don't have one recorded.*

Computational cost

Computing the gradient requires one adjoint solve.

One line of code for efficiently computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, mc, project=True)
```

Dolfin-adjoint derives and solves the adjoint equations for us and returns the gradient.

Note

If you call **compute_gradient** more than once, you need to pass *forget=False* as a parameter. Otherwise you get an error: *Need a value for u_1:0:0:Forward, but don't have one recorded.*

Computational cost

Computing the gradient requires one adjoint solve.

One line of code for efficiently computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, mc, project=True)
```

Dolfin-adjoint derives and solves the adjoint equations for us and returns the gradient.

Note

If you call **compute_gradient** more than once, you need to pass *forget=False* as a parameter. Otherwise you get an error: *Need a value for u_1:0:0:Forward, but don't have one recorded.*

Computational cost

Computing the gradient requires one adjoint solve.

One line of code for efficiently computing Hessians

Dolfin-adjoint can also compute the second derivatives (Hessians):

```
H = hessian(J, mc)
direction = interpolate(Constant(1), V)
plot(H(direction))
```

Computational cost

Computing the directional second derivative requires one tangent linear and two adjoint solves.

One line of code for efficiently computing Hessians

Dolfin-adjoint can also compute the second derivatives (Hessians):

```
H = hessian(J, mc)
direction = interpolate(Constant(1), V)
plot(H(direction))
```

Computational cost

Computing the directional second derivative requires one tangent linear and two adjoint solves.

Verification: How can you check that the gradient is correct?

Taylor expansion of the reduced functional R in a perturbation δm yields:

$$|R(m + \epsilon\delta m) - R(m)| \to 0 \quad \text{at } \mathcal{O}(\epsilon)$$

but

$$|R(m + \epsilon\delta m) - R(m) - \epsilon\nabla R \cdot \delta m| \to 0 \quad \text{at } \mathcal{O}(\epsilon^2)$$

Taylor test

Choose $m, \delta m$ and determine the convergence rate by reducing ϵ. If the convergence order with gradient is ≈ 2, your gradient is probably correct.

```
R = ReducedFunctional(J, mc)
R.taylor_test(m)
```

Dolfin-adjoint Exercise 1

1. Install Dolfin-adjoint
2. Compute the gradient and Hessian of the Poisson example with respect to m.
3. Run the Taylor test to check that the gradient is correct.
4. Measure the computation time for the forward, gradient and Hessian computation. What do you observe? Hint: Use `help(Timer)`.

Notebook tip

Dolfin-adjoint and Notebooks are somewhat orthogonal. If mysterious messages appear, try 'Kernel - Restart & Run All'