FEniCS Course

Lecture 7: Introduction to dolfin-adjoint

Contributors
Simon Funke
Patrick Farrell

What is dolfin-adjoint?

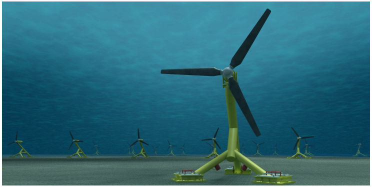
Dolfin-adjoint is FEniCS extoension for: solving adjoint and tangent linear equations; generalised stability analysis; PDE-constrained optimisation.

Main features

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

What has dolfin-adjoint been used for?

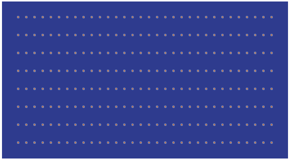
Layout optimisation of tidal turbines

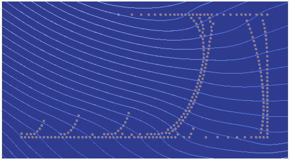


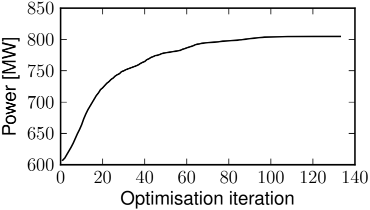
- Up to 400 tidal turbines in one farm.
- What are the optimal locations to maximise power production?

What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines







What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines

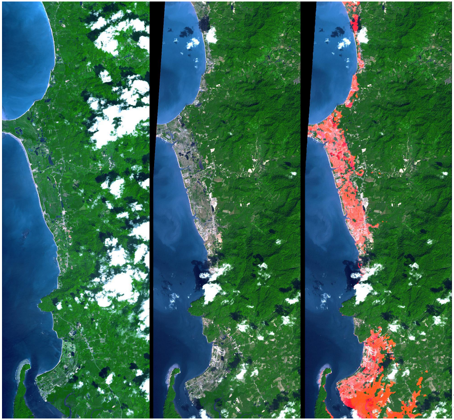
```
from dolfin import *
from dolfin_adjoint import *

# FEniCS model
# ...

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

What has dolfin-adjoint been used for?

Reconstruction of a tsunami wave

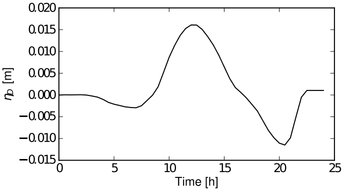
[1]

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

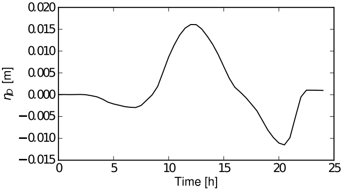
[1] Image: ASTER/NASA PIA06671

What has dolfin-adjoint been used for?

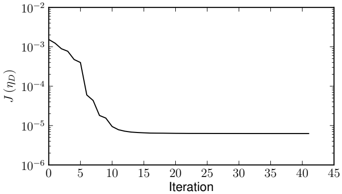
Reconstruction of a tsunami wave



Correct tsunami wave



Reconstructed tsunami wave



Reconstruction of a tsunami wave

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

# FEniCS model
# ...

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

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
- …

Example

Compute the sensitivity of

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

with known u_d and the Poisson equation:

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

with respect to f and ν.

Poisson solver in FEniCS

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

```
from dolfin import *
mesh = UnitSquareMesh(50, 50)
V = FunctionSpace(mesh, "CG", 1)

# Define Functions
u = TrialFunction(V)
v = TestFunction(V)
s = Function(V)
f = interpolate(Constant(1), V)
nu = Constant(1)

# Define variational forms
a = nu*inner(grad(u), grad(v))*dx
L = f*v*dx

# Solve problem
bcs = DirichletBC(V, 0.0, "on_boundary")
solve(a == L, s, bcs)
```

Dolfin-adjoint (i): Annotation

The first change necessary to adjoin this code is to import the dolin-adjoint module *after* loading dolfin:

```
from dolfin 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 (ii): Objective functional

Next, we implement the objective functional, the square of the norm of u

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

or in code

```
# ...
J = Functional(inner(s-ud, s-ud)*dx)
```

Dolfin-adjoint (ii): Parameter

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 f, hence we use:

```
m = SteadyParameter(f)
```

Other Parameters are availabe. The most common are:

- **SteadyParameter**: For steady state problems.
- **InitialConditionParameter**: For the initial condition of time-dependent problems.
- **ScalarParameter**: For **Constant** parameters.

Dolfin-adjoint (iii): Computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, m, 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.

Dolfin-adjoint (iii): Computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, m, 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.

Dolfin-adjoint (iii): Computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, m, 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.

Dolfin-adjoint (iii): Computing Hessians

Dolfin-adjoint can also compute the second derivatives:

```
hess = hessian(J, m)
direction = interpolate(Constant(1), V)
plot(hess(direction))
```

Computational cost

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

Dolfin-adjoint (iii): Computing Hessians

Dolfin-adjoint can also compute the second derivatives:

```
hess = hessian(J, m)
direction = interpolate(Constant(1), V)
plot(hess(direction))
```

Computational cost

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

Dolfin-adjoint (iii): Time-dependent problems

For time-depedent problems, you need to tell dolfin-adjoint when a new time-step starts:

```
# Set the initial time
adjointer.time.start(t)
while (t <= end):
    # ...
    # Update the time
    adj_inc_timestep(time=t, finished=t>end)
```

Time integration

Dolfin-adjoint adds the time measure **dt** which you can use to integrate a functional over time. Examples:

```
J1 = Functional(inner(s, s)*dx*dt)
J2 = Functional(inner(s, s)*dx*dt[FINISH_TIME])
J3 = Functional(inner(s, s)*dx*dt[0.5])
J4 = Functional(inner(s, s)*dx*dt[0.5:])
```

Verification

How can you check that the gradient is correct?

Taylor expansion of the reduced functional $\tilde{J}$ in a perturbation δm yields:

$$|\tilde{J}(m + \epsilon \delta m) - J(m)| \to 0 \quad \text{at } \mathcal{O}(\epsilon) \tag{1}$$

but

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

Tayor test

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

The function **help(taylor_test)** implements the Taylor test for you.

Verification

How can you check that the gradient is correct?

Taylor expansion of the reduced functional $\tilde{J}$ in a perturbation δm yields:

$$|\tilde{J}(m + \epsilon\delta m) - J(m)| \to 0 \quad \text{at } \mathcal{O}(\epsilon) \tag{1}$$

but

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

Tayor test

Choose $m, \delta m$ and determine the convergence rate by reducing ϵ. If the convergence order with gradient is ≈ 2, your gradient is correct.
The function **help(taylor_test)** implements the Taylor test for you.

The FEniCS challenge!

1. Compute the gradient and Hessian of the Poisson example with respect to ν and f. Do you get the same gradient as yesterday? Hint: you can pass a list of parameters to *compute_gradient*.
2. Measure the computation time for the forward, gradient and Hessian computation. Hint: Use *help(Timer)*. What do you observe?
3. Solve the Burger's equation

$$\frac{\partial u}{\partial t} - \nu \Delta u + u \cdot \nabla u = 0,$$

and compute the gradient of $J(u) = \int_\Omega \|u\| \,\mathrm{d}x \,\mathrm{d}t$ with respect to the initial condition. Time the forward and gradient computation. Which one is faster. Why?