FEniCS Course

Lecture 13: Introduction to dolfin-adjoint

Contributors
Simon Funke
Patrick Farrell
Marie E. Rognes

Often we are interested how sensitiv a model output is with respect it the model inputs

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*

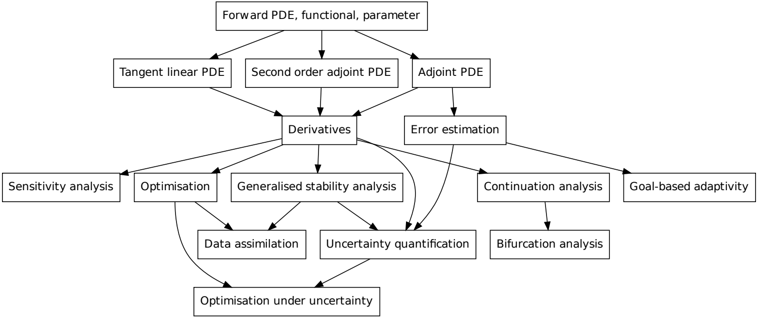
$$J(u) = \frac{1}{2} \int_\Omega \|u - u_d\|^2 \, \mathrm{d}x,$$

where u_d is a known function.

Goal

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

Sensitivities are ubiquitous in scientific computing



Comput. deriv. (i) General formulation

Given

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

Goal

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

Comput. deriv. (i) General formulation

Given

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

Goal

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

Adjoint approach

1. Solve the adjoint equation for λ

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

2. Compute

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

Deriving and implementating the adjoint is challenging

From "The Art of Differentiating Computer Programs" (Naumann, 2011):

> *[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.*

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`.

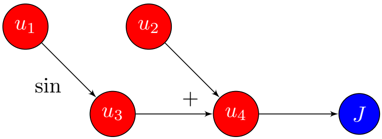
The traditional approach to deriving discrete adjoints

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

Algorithmic differentiation

Fundamental idea of AD

A model is a sequence of elementary instructions.



Difficulty

- pointers
- aliasing
- expressions with side effects
- preprocessor directives
- memory allocation
- external libraries
- mixed-language programming
- parallel directives

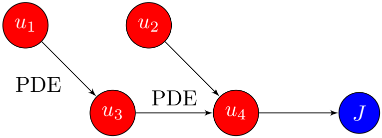
The approach in `dolfin-adjoint`

$$\begin{tikzcd}
\boxed{\text{discrete forward equations}} \arrow[r, "\text{code generation (FEniCS)}"] \arrow[d, "\text{symbolic manipulation (dolfin-adjoint)}"] & \boxed{\text{forward code}} \\
\boxed{\text{discrete adjoint equations}} \arrow[r, "\text{code generation (FEniCS)}"] & \boxed{\text{adjoint code}}
\end{tikzcd}$$

This approach

Fundamental idea of this work

A (finite element) model is a sequence of PDE solves.



The tape is *a record of the equations solved*. Same as AD, but much higher level.

Symbolic representation in FeniCS

Burgers' equation (strong)

$$F = u \cdot \nabla u - \nu \nabla^2 u - f = 0$$

Burgers' equation (weak)

$$F = (u \cdot \nabla u, v) + \nu \left(\nabla u, \nabla v\right) - (f, v) = 0$$

Burgers' equation (code)

```
F = (u*grad(u)*v + nu*grad(u)*grad(v)- f*v)*dx == 0
```

Symbolic representation in FeniCS

Burgers' equation (strong)

$$F = u \cdot \nabla u - \nu \nabla^2 u - f = 0$$

Burgers' equation (weak)

$$F = (u \cdot \nabla u, v) + \nu \, (\nabla u, \nabla v) - (f, v) = 0$$

Burgers' equation (code)

```
F = (u*grad(u)*v + nu*grad(u)*grad(v)- f*v)*dx == 0
```

Symbolic representation in FeniCS

Burgers' equation (strong)

$$F = u \cdot \nabla u - \nu \nabla^2 u - f = 0$$

Burgers' equation (weak)

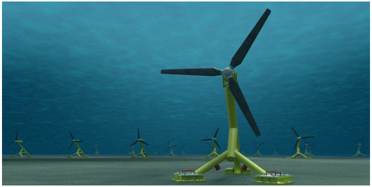
$$F = (u \cdot \nabla u, v) + \nu\,(\nabla u, \nabla v) - (f, v) = 0$$

Burgers' equation (code)

```
F = (u*grad(u)*v + nu*grad(u)*grad(v)- f*v)*dx == 0
```

What has dolfin-adjoint been used for?

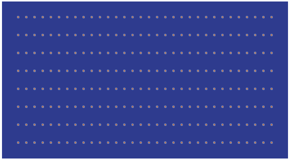
Layout optimisation of tidal turbines

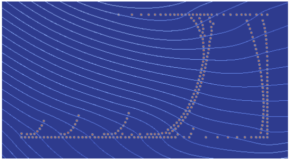


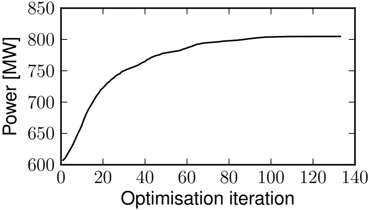
- Up to 400 tidal turbines in one farm.
- What are the optimal locations to maximise power production?

What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines







What has dolfin-adjoint been used for?

Layout optimisation of tidal turbines

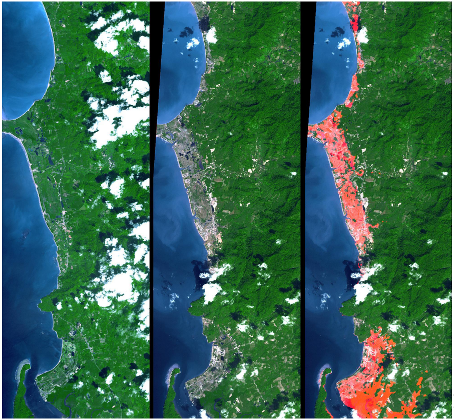
```
from dolfin import *
from dolfin_adjoint import *

# 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

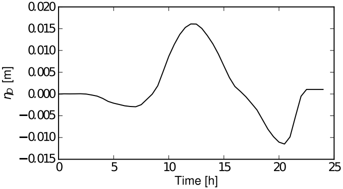
[1]

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

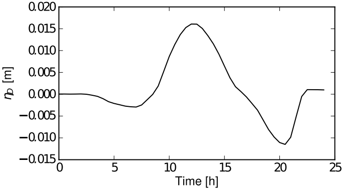
[1]Image: ASTER/NASA PIA06671

What has dolfin-adjoint been used for?

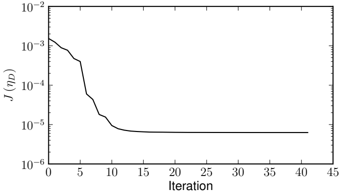
Reconstruction of a tsunami wave



Correct tsunami wave



Reconstructed tsunami wave



Reconstruction of a tsunami wave

```python
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
- ...

Example

Compute the sensitivity of

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

with known u_d and the Poisson equation:

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

with respect to m.

Poisson solver in FEniCS

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

```python
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 (i): Annotation

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 (ii): Objective Functional

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 = assemble(j)
```

Dolfin-adjoint (ii): Control 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 m.

We inform dolfin-adjoint of this:

```
ctrl = Control(m)
```

Dolfin-adjoint (iii): Computing gradients

Now, we can compute the gradient with:

```
dJdm = compute_gradient(J, ctrl,
    options={"riesz_representation": "L2"})
```

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

Note

The derivative is stored as its Riesz representation. The Riesz representation depends on the inner product (here we chose L2). Other supported inner products are l2 and H1.

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, ctrl,
    options={"riesz_representation": "L2"})
```

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

Note

The derivative is stored as its Riesz representation. The Riesz representation depends on the inner product (here we chose L2). Other supported inner products are l2 and H1.

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, ctrl,
    options={"riesz_representation": "L2"})
```

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

Note

The derivative is stored as its Riesz representation. The Riesz representation depends on the inner product (here we chose L2). Other supported inner products are l2 and H1.

Computational cost

Computing the gradient requires one adjoint solve.

Dolfin-adjoint (iii): Computing Hessians

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

```
H = hessian(J, m)
direction = interpolate(Constant(1), V)
plot(H(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 (Hessians):

```
H = hessian(J, m)
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.

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

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.

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

Getting started with Dolfin-adjoint

1. Install dolfin-adjoint (see `dolfin-adjoint.org`)
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 (Hint: you need to create a 'ReducedFunctional' object).
4. Measure the computation time for the forward, gradient and Hessian computation. What do you observe? Hint: Use `dolfin.Timer`.