PDE-constrained optimisation using automated adjoints of finite element models.

S. W. Funke, P. E. Farrell, D. A. Ham, M. E. Rognes, M. D. Piggott

Imperial College London
Simula Research Laboratory

December 17, 2012

What is PDE-constraint optimisation?

General form

$$\min_m J(u,m)$$

subject to

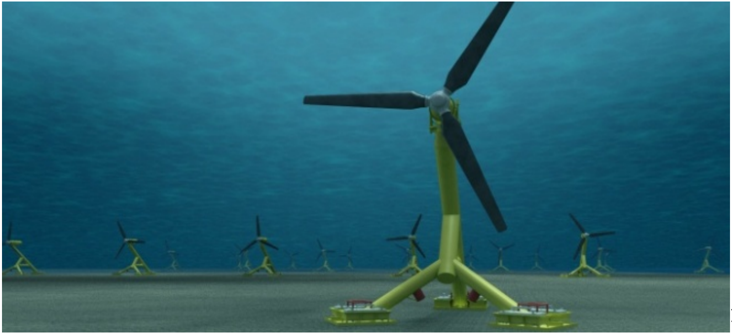
$$F(u,m) = 0$$

where

- m is a vector containing the *optimisation variables*
- $J \in \mathbb{R}$ is the *functional of interest*
- F is a *partial differential equation* (PDE) with solution vector u.

Example problems

What is the turbine array layout that extracts most energy from a tidal current?


[1]

[1]Image credit: Hammerfest Strom AS

Example problems

What is the turbine array layout that extracts most energy from a tidal current?

Optimal turbine layout

$$\max_m \mathsf{Power}(u, m)$$

subject to

$$u_t + \nabla \eta = -s(u, m),$$
$$\eta_t + \nabla \cdot u = 0.$$

m: turbine positions,
u: velocity,
η: water elevation.

Example problems

Optimal control of the heat equation

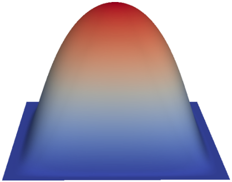
$$\min_{m} \ \frac{1}{2}||u-d||^2 + \frac{\alpha}{2}||m||^2$$

subject to

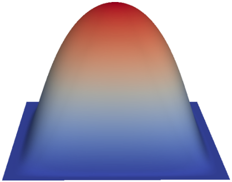
$$\begin{aligned} \nabla^2 u &= m && \text{on } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned}$$

m: heat source,
u: temperature profile,
d: desired temperature profile.

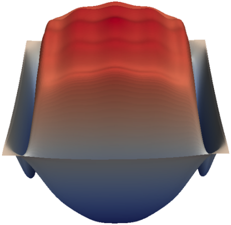
Optimal control of the heat equation



desired



optimal temperature profile



optimal heat source

Applications

Many more examples of PDE-constrained optimisation problems in research and industry:

- Design optimisation
- Data assimilation
- Parameter estimation
- Optimal control

What this talk is about

A high-level framework for PDE-constrained optimisation problems

- Minimal development effort for the user
- High level input language that resembles the mathematical structure
- Usage of gradient based optimisation algorithms
- Efficiency through code generation/parallel execution

The result of this talk will be the optimal heat control problem in less than 30 lines of code.

Back to the basics

$$\min_m J(u, m)$$
$$\text{s.t. } F(u, m) = 0$$

Typical optimisation loop

- Choose initial m
- **do**
  - Compute functional J by solving the forward PDE
  - (Compute the gradient dJ/dm) by solving the adjoint PDE
  - Update optimisation variables m
- **while** not converged

PDE environment

The FEniCS project

- A problem solving environment for finite element discretisations
- High level Python interface with domain specific language
- Just in time compiler produces highly optimised C++ code

Symbolic representation

With a symbolic representation of the discretisation, it is possible to *automate the process of implementing finite element models*.

Burgers' equation (maths)

$$F = \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u = 0$$

Symbolic representation

With a symbolic representation of the discretisation, it is possible to *automate the process of implementing finite element models*.

Burgers' equation (maths)

$$F = \frac{\partial u}{\partial t} + u \cdot \nabla u - \nu \nabla^2 u = 0$$

Burgers' equation (code)

```
F = ((u - u_old)/dt*v + u*grad(u)*v + nu*grad(u)*grad(v))*dx
```

Demonstration

```
""" Solves the heat equation """
from dolfin import *

# Define domain
n = 200
mesh = Rectangle(-1, -1, 1, 1, n, n)
V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="State")
v = TestFunction(V)

# Set  source term value
m = project(Expression("x[0]*x[1]"), V)

# Solve the PDE
F = (inner(grad(u), grad(v)) - m*v)*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)
```

The code for the heat equation

Adjoint equation

Given a PDE $F(u,m) = 0$ and a functional $J(u,m)$, the adjoint equation is:

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

Key properties

- The adjoint equation is linear and depends on u.
- The adjoint equation is solved backward in time (upper-triangular)
- The functional gradient is obtained by solving

$$\frac{dJ}{dm} = -\lambda^* \frac{\partial F}{\partial m} + \frac{\partial J}{\partial m}.$$

Hence the derivative computation requires **one** forward solve for u and **one** adjoint solve for λ, independent of the size of m.

The problem

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.*

This is the problem we're trying to solve (for finite elements, at least).

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

A model is a sequence of elementary numerical instructions. Differentiate each in turn, and compose with the chain rule.

Algorithmic differentiation

Fundamental idea

A model is a sequence of elementary numerical instructions. Differentiate each in turn, and compose with the chain rule.

Drawbacks

- Major investment of labour ("semi-automatic")
- Does not naturally work in parallel (manual intervention)
- Adjoint can be very slow (Naumann (2011): 3–30$\times$ slower)
- Checkpointing requires large amount of intervention

This makes differentiating code very hard.

The approach in `dolfin-adjoint`

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

The abstractions

Libadjoint

A model is a sequence of equation solves. Differentiate each in turn, and compose with the chain rule.

FEniCS

The equations to be solved are represented as data. Use a compiler to generate implementation details.

Demonstration

```
""" Solves the heat equation """
from dolfin import *
from dolfin_adjoint import *

# Define domain
n = 200
mesh = Rectangle(-1, -1, 1, 1, n, n)
V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="State")
v = TestFunction(V)

# Set  source term value
m = project(Expression("x[0]*x[1]"), V)

# Solve the PDE
F = (inner(grad(u), grad(v)) - m*v)*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)

# Define the desired temperature profile
x = triangle.x
d = exp(-1/(1-x[0]*x[0])-1/(1-x[1]*x[1]))

# Define the functional of interest
J = Functional((0.5*inner(u-d, u-d))*dx*dt[FINISH_TIME])

# Compute the gradient with the adjoint approach
dJdm = compute_gradient(J, InitialConditionParameter(m))
```

Adjoint code for the heat equation

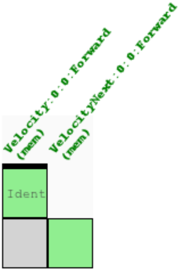
Building the tape

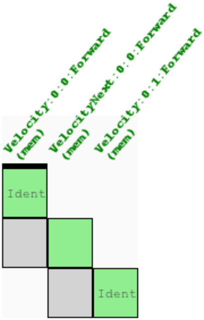
The tape is a record of the equations solved.

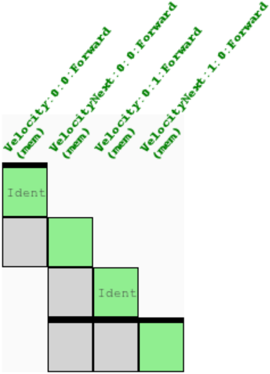
Operator overloading

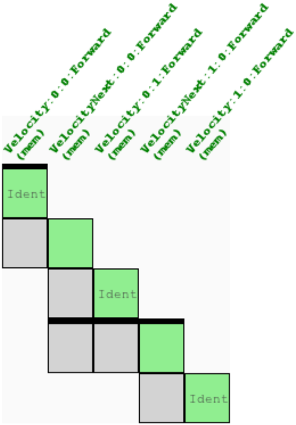
Overload *functions that create new values*:

- `solve`
- `assign`
- ...


Velocity:0:0:Forward (mem)
VelocityNext:0:0:Forward (mem)
Ident


Velocity:0:0:Forward
(mem)
VelocityNext:0:0:Forward
(mem)
Velocity:0:1:Forward
(mem)
Ident
Ident


Velocity:0:0:Forward (mem)
VelocityNext:0:0:Forward (mem)
Velocity:0:1:Forward (mem)
VelocityNext:1:0:Forward (mem)
Ident
Ident


Velocity:0:0:Forward (mem)
VelocityNext:0:0:Forward (mem)
Velocity:0:1:Forward (mem)
VelocityNext:1:0:Forward (mem)
Velocity:1:0:Forward (mem)
Ident
Ident
Ident

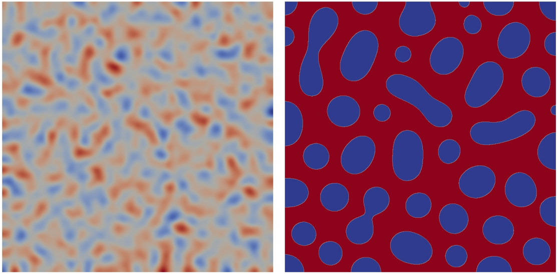
Velocity:0:0:Adjoint
VelocityNext:0:0:Adjoint
Velocity:0:1:Adjoint
VelocityNext:1:0:Adjoint
Velocity:1:0:Adjoint
Ident
7130c
Ident
73451
Ident

Advantages

Advantages

- The adjoint derivation is totally automatic
- The adjoint is very efficient
- The adjoint can automatically use checkpointing (`revolve`)
- The adjoint works naturally in parallel (both MPI and OpenMP)

Adjoint computation: Cahn-Hilliard



(a) $t = 0.0$ (b) $t = 2.5 \times 10^{-4}$

Quite a difficult problem to adjoint:

- nonlinear, time-dependent, fourth-order equation
- run in parallel (8 processors, MPI)
- checkpointing necessary (5 in memory, 10 on disk)

Adjoint computation: Cahn-Hilliard

Willmore functional:

$$W(u(t)) = \frac{1}{4\epsilon} \int_\Omega \left(\epsilon \nabla^2 u(t = T) - \frac{1}{\epsilon} \frac{df}{dc} \right)^2 \; dx$$

which is intimately connected to the finite-time stability of transition solutions of the Cahn-Hilliard equation.

Adjoint computation: Cahn-Hilliard

`dolfin-adjoint` gets the **correct** adjoint:

h	$\hat{W}(\tilde{u_0}) - \hat{W}(u_0)$	order	$\hat{W}(\tilde{u_0}) - \hat{W}(u_0) - \tilde{u_0}^T \nabla \hat{W}$	order
1×10^{-5}	14.6197		0.5680	
5×10^{-6}	7.4485	0.9728	0.14532	**1.9667**
2.5×10^{-6}	3.7602	0.9861	0.03666	**1.9869**
1.25×10^{-6}	1.8892	0.9930	0.009202	**1.9941**
6.25×10^{-7}	0.9469	0.9964	0.002304	**1.9972**

Table: The Taylor remainders for the Willmore functional $\hat{W}$. If the orders are 2, the adjoint is correct.

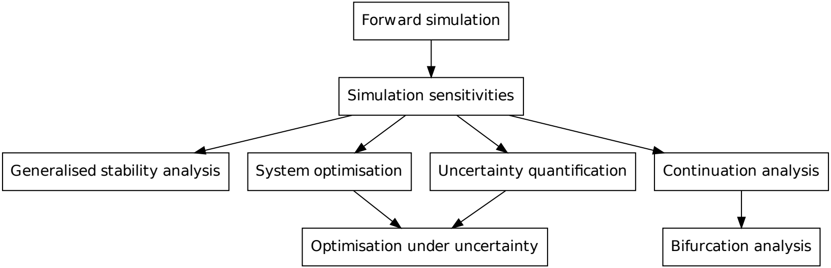
Adjoint computation: Cahn-Hilliard

`dolfin-adjoint` gets the adjoint **quickly**:

	Runtime (s)	Ratio
Forward model	103.93	
Forward model + annotation	104.24	**1.002**
Forward model + annotation + adjoint model	127.07	**1.22**

Table: Timings for the Cahn-Hilliard adjoint (without checkpointing). The overhead of annotation is less than 1%. The adjoint model takes approximately 22% of the cost of the forward model.

Applications



Revisiting the optimisation loop

Optimisation loop

- Choose initial m
- **do**
  - Compute functional J
  - (Compute the gradient dJ/dm)
    by solving the adjoint PDE with *dolfin-adjoint*
  - Update optimisation variables m
- **while** not converged

Main idea of *dolfin-adoint.optimize*

Perform the optimisation steps purely on the tape.

Replay the tape

The tape of *dolfin-adjoint* can not only be used to derive and solve the adjoint PDE but also to recompute the forward PDE and the functional value.

Update the tape

Every time the optimisation algorithm computes new values for the optimisation variables, update the tape accordingly.

Main idea of *dolfin-adoint.optimize*

The optimisation loop

- Choose initial m
- **do**
  - Compute functional J
    by replaying *dolfin-adjoint*'s tape
  - (Compute the gradient dJ/dm)
    by solving the adjoint PDE with *dolfin-adjoint*
  - Update optimisation variables m
    by updating *dolfin-adjoint*'s tape
- **while** not converged

User interface

The result is an extremely compact user interface:

```
# ... create tape by solving the forward PDE once
rf = ReducedFunctional(J, m)
m_opt = minimize(rf)
```

Demonstration

```
""" Solves the mother problem in optimal control of PDEs
from dolfin import *
from dolfin_adjoint import *
dolfin.set_log_level(ERROR)

# Define domain
n = 200
mesh = Rectangle(-1, -1, 1, 1, n, n)
V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="State")
m = Function(V, name="Control")
v = TestFunction(V)

# Define the PDE
F = (inner(grad(u), grad(v)) - m*v)*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)

# Define the desired temperature profile
x = triangle.x
d = exp(-1/(1-x[0]*x[0])-1/(1-x[1]*x[1]))

# Define the functional of interest
J = Functional((0.5*inner(u-d, u-d))*dx*dt[FINISH_TIME])

# Run the optimisation
rf = ReducedFunctional(J, InitialConditionParameter(m))
m_opt = minimize(rf, pgtol=2e-08, iprint = 1)
```

Code for solving the optimal control problem of the heat equation

Key features

Currently interfaces to:

- Sequential least squares programming
- BFGS
- L-BFGS-B
- Truncated Newton algorithm

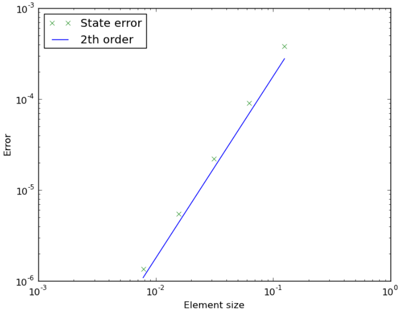
Additional constraints are supported (depending on the optimisation algorithm):

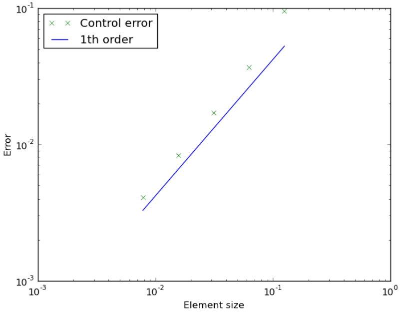
- Control bounds
- (In-)Equality constraints

Analytical convergence test

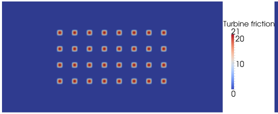
Test is based on an analytical solution to the optimal control of the heat equation. The expected convergence rates are:

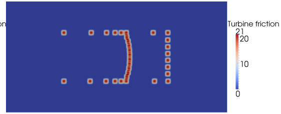
- State: 2nd order (discretised with P1-elements)
- Control: 1st order (discretised with discontinuous P0-elements)

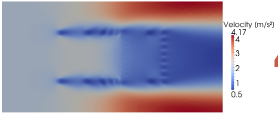


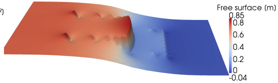


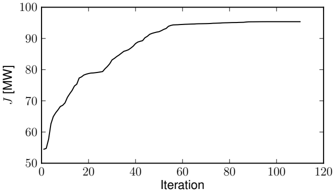
Optimal placement of tidal turbines



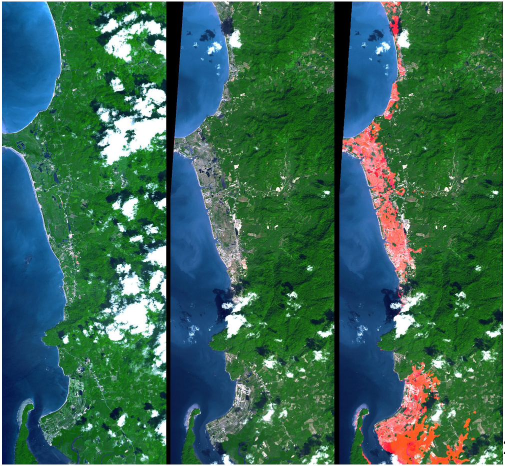








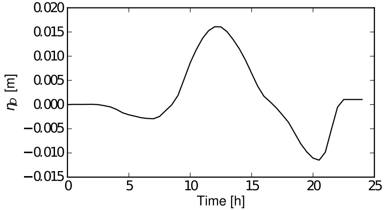
Reconstruction of a tsunami wave


[2]

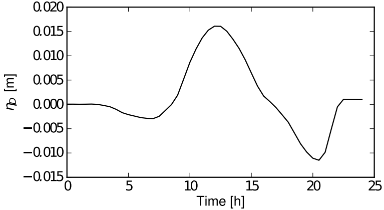
Is it possible to reconstruct the tsunami wave from such images?

[2] Image: ASTER/NASA

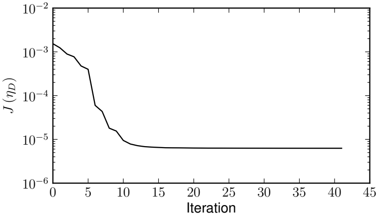
Reconstruction of a tsunami wave



Correct tsunami wave



Reconstructed tsunami wave



Conclusion

- The aim is to develop a framework for rapidly solving PDE-constrained optimisation problems.
- The code is under heavy development, but works for many useful cases.

Future work includes:

- Multi-objective optimisation
- Support for shape optimisation

`dolfin-adjoint`

http://dolfin-adjoint.org