DOLFIN-ADJOINT
AUTOMATIC ADJOINT MODELS FOR FENICS

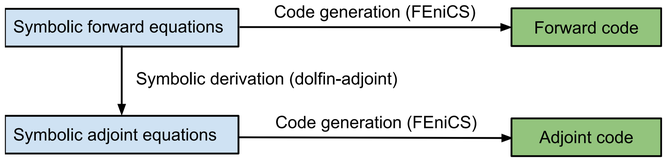
P. E. Farrell, S. W. Funke, D. A. Ham, M. E. Rognes
Simon@simula.no

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.

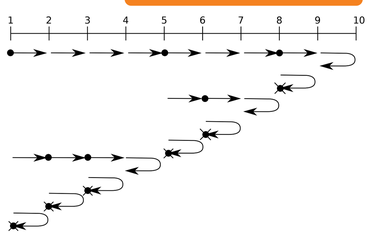


Figure: Visualisation of the optimal checkpointing strategy with 10 time levels and 3 checkpoints

To benchmark the checkpoint implementation, we used the sensitivity example to compare the additional computational cost of checkpointing with a store-all strategy in dolfin-adjoint:

Slow-down factor with 11 timesteps and varying memory checkpoints

Theoretical adjoint to forward runtime ratio	5.00	2.18	1.63	1.45	1.00
Observed adjoint to forward runtime ratio	5.07	2.26	1.73	1.53	0.90

How to get started

http://dolfin-adjoint.org

Contains introduction to adjoints, a tutorial, examples, and installation instructions for Linux with Ubuntu packages and MacOS X.

Downloads

This poster PDF format

References dolfin-adjoint.org/citing

Source code bitbucket.org/dolfin-adjoint

Application examples

Sensitivity analysis

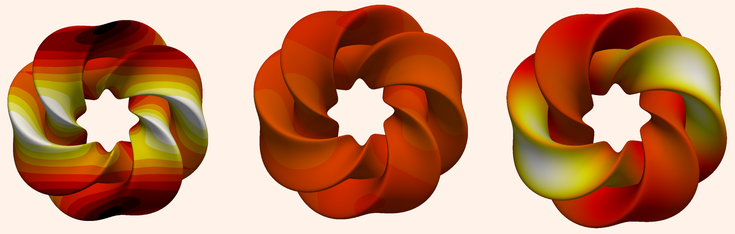
Consider the time dependent heat equation

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

Here Ω is the Gray's Klein bottle, a closed 2D manifold embedded in 3D, T is the final time, u is the unkown 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 minimises the compliance

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

subject to the Poisson equation with mixed Dirichlet–Neumann conditions

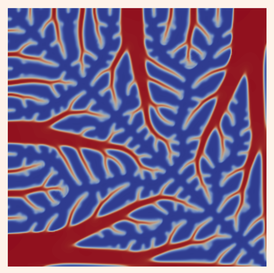
$$-\mathrm{div}(k(a)\nabla T) = f \qquad \text{in } \Omega,$$
$$T = 0 \qquad \text{on } \partial\Omega_D,$$
$$k(a)\nabla T = 0 \qquad \text{on } \partial\Omega_N,$$

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 parameterisation, α is a regularisation term, and V is the volume bound on the control. Physically, the problem is to find the material distribution a that minimises 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

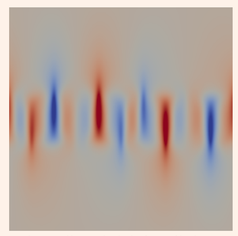
This example performs a generalized stability analysis to find the perturbations to an initial condition that grow the most over some finite time. The governing equations are the two-dimensional vorticity-streamfunction formulation of the time-dependent Navier–Stokes equations, coupled to two advection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

$$\frac{\partial \zeta}{\partial t} + \nabla^\perp \psi \cdot \nabla \zeta = \frac{\mathrm{Ra}}{\mathrm{Pr}} \left(\frac{\partial T}{\partial x} - \frac{1}{R_\rho^0} \frac{\partial S}{\partial x} \right) + \nabla^2 \zeta,$$
$$\frac{\partial T}{\partial t} + \nabla^\perp \psi \cdot \nabla T = \frac{1}{\mathrm{Pr}} \nabla^2 T,$$
$$\frac{\partial S}{\partial t} + \nabla^\perp \psi \cdot \nabla S = \frac{1}{\mathrm{Sc}} \nabla^2 S,$$
$$\nabla^2 \psi = \zeta.$$

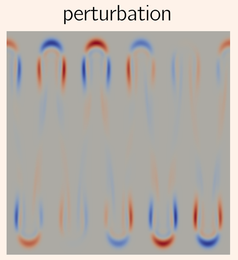
ζ is the vorticity, ψ is the streamfunction, T is the temperature, S is the salinity, and Ra, Sc, Pr and R_ρ^0 are parameters. The configuration consists of two well-mixed layers (i.e., of homogeneous temperature and salinity) separated by an interface. The instability is activated by a sinusoidal perturbation to the initial salinity field.

```
from dolfin import *
from dolfin_adjoint import *
# ...
gst = compute_gst("InitialSalinity", "FinalSalinity", nsv=2)
```

Initial salinity



Leading initial salinity perturbation



Leading final salinity perturbation

◁ **Code**: Implementation excerpt (the full code uses SLEPc and has 144 lines)

Center for Biomedical Computing
EPSRC Engineering and Physical Sciences Research Council
NERC SCIENCE OF THE ENVIRONMENT