An algorithmic differentiation tool (not only) for FEniCS

Sebastian Mitusch and Simon W. Funke

Automatically derive and solve adjoint and tangent linear equations from FEniCS models

Highlights

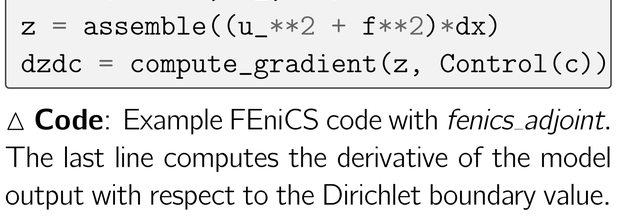
- Provides algorithmic-differentiation (AD) in FEniCS (extendable to other frameworks).
- Computes gradients, directional derivatives, and Hessian actions of model outputs with minimal code changes.
- Natural parallel-support and close-to-theoretical performance in FEniCS.
- Currently being extended to PyTorch to enable coupling FEniCS and PyTorch models.

How it works

The implementation consists of two modules:

pyadjoint

A generic, operator-overloading AD tool for Python. During runtime, *pyadjoint* records all overloaded operations, and their inputs/outputs (as *Variables*) as a graph. From this graph, the derivatives of any node with respect to any preceding node can be computed by successive application of the chain rule.



◁ **Figure**: The forward model registers each operation and its inputs/outputs as a graph. The stored *Operation* objects can evaluate the operation for new inputs, evaluate the tangent linear, and the first or the second-order adjoint operations.

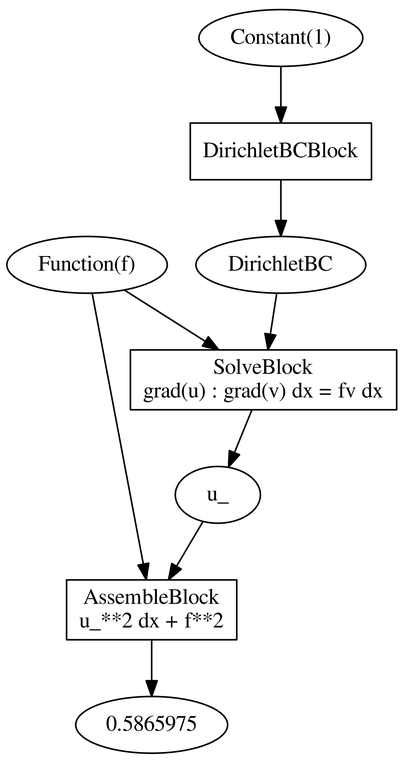
fenics_adjoint

This module overloads the most common FEniCS operations.

```
from fenics import *
from fenics_adjoint import *
# ...
c = Constant(1)
bc = DirichletBC(V, c,
                 "on_boundary")
a = inner(grad(u), grad(v))*dx
L = f*v*dx
solve(a == L, u_, bc)
z = assemble((u_**2 + f**2)*dx)
dzdc = compute_gradient(z, Control(c))
```

△ **Code**: Example FEniCS code with *fenics_adjoint*. The last line computes the derivative of the model output with respect to the Dirichlet boundary value.

Figure: ▷
Visualisation of the recorded *pyadjoint* computation graph after executing the above code. The main high-level FEniCS operations have been recorded. The output variable z is also overloaded and could further used, for example to evaluate a more complex functional.



Performance

The adjoint and tangent linear models inherits the parallelism and scalability of FEniCS.

Time-dependent example

CPUs	1	Optimal
Forward runtime (s)	1.34	
Adjoint runtime (s)	0.68	
Adjoint/Forward ratio	0.51	0.33

Inverse problem example

CPUs	1	2	Optimal
Forward runtime (s)	84.2	35.2	
Adjoint runtime (s)	106.4	47.0	
Adjoint/Forward ratio	1.26	1.34	1.00

Tables: Performance timings for the two examples on the right.

Time-dependent functional

In this example we consider Burger's equation

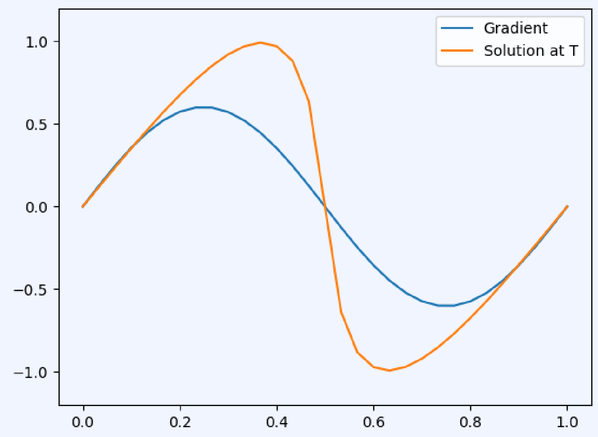
$$\frac{\partial u}{\partial t} + \alpha u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad \text{in } \Omega \times (0, T),$$

$$u = g \qquad \text{for } \Omega \times \{0\}.$$

Here Ω is the unit interval, and $T = 0.3$. The aim is to compute the sensitivity of the functional

$$J(u) = \int_0^T \int_\Omega u^2 \, dx \, dt$$

with respect to the initial condition g and the constant α.




```
from fenics import *
from fenics_adjoint import *

F = ((u-u_)/dt*v
+ a*u*u.dx(0)*v
+ nu*u.dx(0)*v.dx(0))*dx
J = 0

u_.assign(g)
while (t <= T):
        solve(F == 0, u, bc)
        u_.assign(u)
        t += timestep
        J += dt*assemble(u**2*dx)

# Apply fenics-adjoint
dJdga = compute_gradient(J, [g,a],
{"riesz_representation": "L2"})
```

△ **Code**: Simplified FEniCs implementation (complete code has 32 lines)
◁ **Figure**: The solution u at T and the L^2-gradient with respect to the initial condition.

Inverse problem

The idea is to use magnetic resonance imaging (MRI) images of patients together with partial differential equation (PDE) constrained optimization, to estimate the apparent diffusion coefficient in the brain extracellular space.
Specifically, we consider the following constrained minimization problem

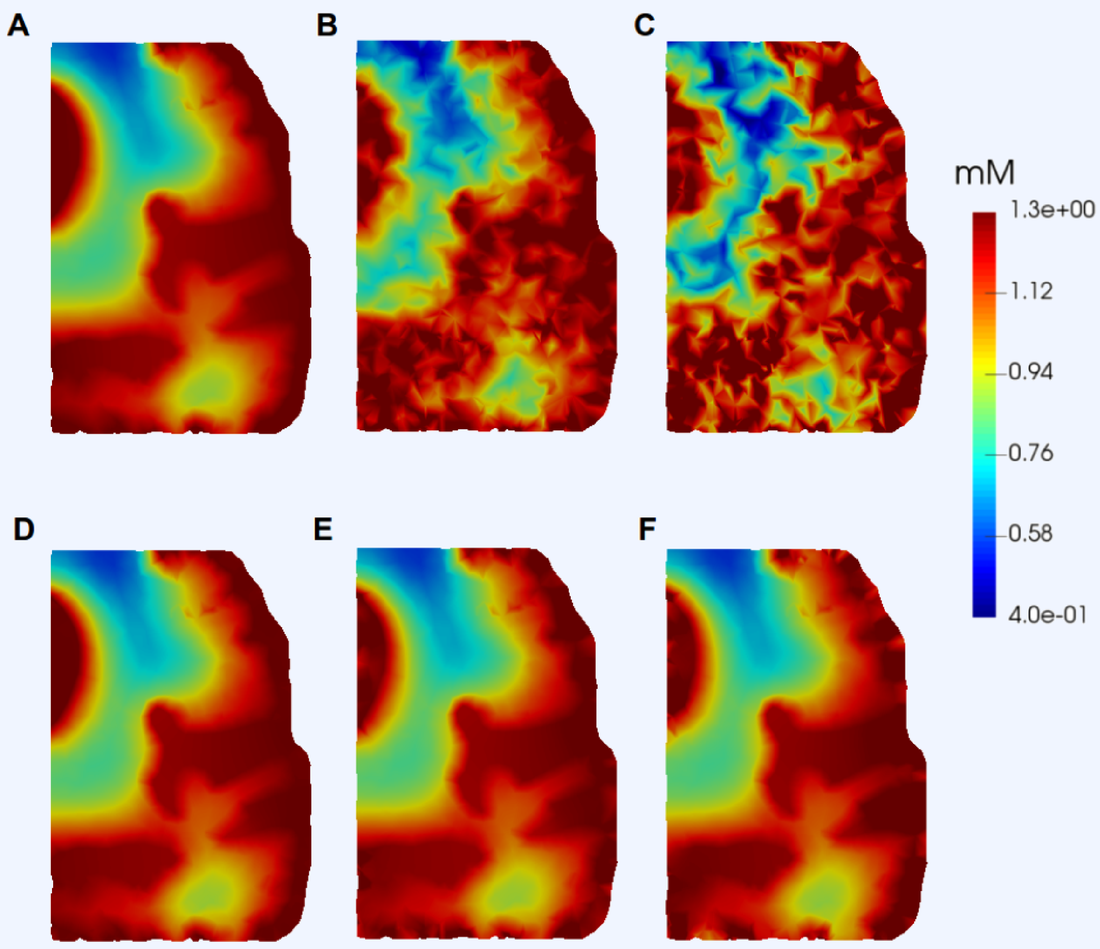
$$\min_{D,g} \sum_{i \in \mathcal{I}_d} \int_\Omega |u(t_i) - u_{obs}(t_i)|^2 \, d\Omega + \mathcal{R}(g)$$

subject to

$$\frac{\partial u}{\partial t} - \nabla \cdot D \nabla u = 0 \quad \text{in } \Omega,$$

$$u = g \quad \text{on } \partial\Omega$$

To test the robustness of the method, we manufacture solutions and add noise to the observations.



△ **Figure**:
The manufactured solution (upper row) versus the estimated solution (lower row) at a single time point. (A) is the manufactured solution without added noise. (B) is the manufactured solution with noise amplitude 0.15. (C) is the manufactured solution with noise amplitude 0.30. (D) shows the estimated solution state using observation (A). (E) shows the estimated solution state using observation (B). (F) shows the estimated solution state using observation (C).

How to get started

pip install git+https://bitbucket.org/dolfin-adjoint/pyadjoint@master