A new 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

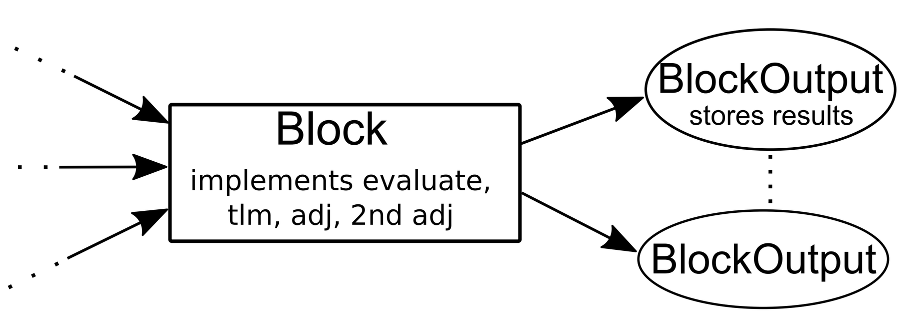
- A new algorithmic-differentiation (AD) tool for FEniCS (extendable to other frameworks).
- Computes gradients, directional derivatives, and Hessian actions of model outputs with minimal code changes.
- Supports *solve*, *project*, *assemble*, *DirichletBC*s, *Expression*, ...
- Natural parallel-support and close-to-theoretical performance.

How it works

The implementation consists of two modules:

pyadjoint

A generic, operator-overloading AD tool for Python. During run-time, *pyadjoint* records all overloaded operations (as *Blocks*), their outputs (as *BlockOutputs*) and their dependencies as a graph. From this graph, the derivatives of any leave node with respect to any root node can be computed by successive application of the chain rule.



◁ **Figure**: The forward model registers each operation as a *Block* and its results as *BlockOutputs*. A *Block* can evaluate the operation for new inputs, evaluate the tangent linear, and the first or the second-order adjoint operations.

fenics_adjoint

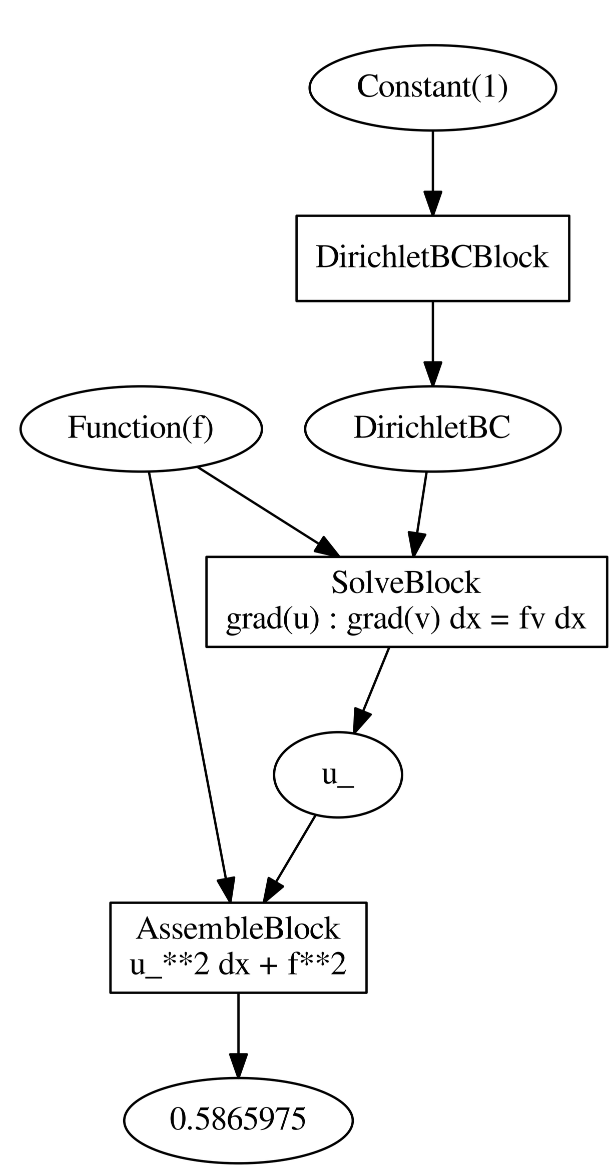
This module overloads the most common FEniCS operations.

```
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, 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.

Dirichlet boundary controls example

CPUs	1	2	Optimal
Forward runtime (s)	13.7	5.59	
Adjoint runtime (s)	16.6	6.53	
Adjoint/Forward ratio	1.21	1.17	1.00

Time-dependent example

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

Tables: Performance timings for the two examples on the right. Similar forward-to-adjoint ratios were observed with *dolfin-adjoint*.

Dirichlet boundary controls

Consider the Stokes equations

$$-\nu\nabla^2 u + \nabla p = f \quad \text{in } \Omega,$$
$$\operatorname{div} u = 0 \quad \text{in } \Omega.$$

with Dirichlet boundary conditions

$$u = g \quad \text{on } \partial\Omega_{\text{circle}}$$
$$u = f \quad \text{on } \partial\Omega_{\text{in}}$$
$$u = 0 \quad \text{on } \partial\Omega_{\text{walls}}$$
$$p = 0 \quad \text{on } \partial\Omega_{\text{out}}$$

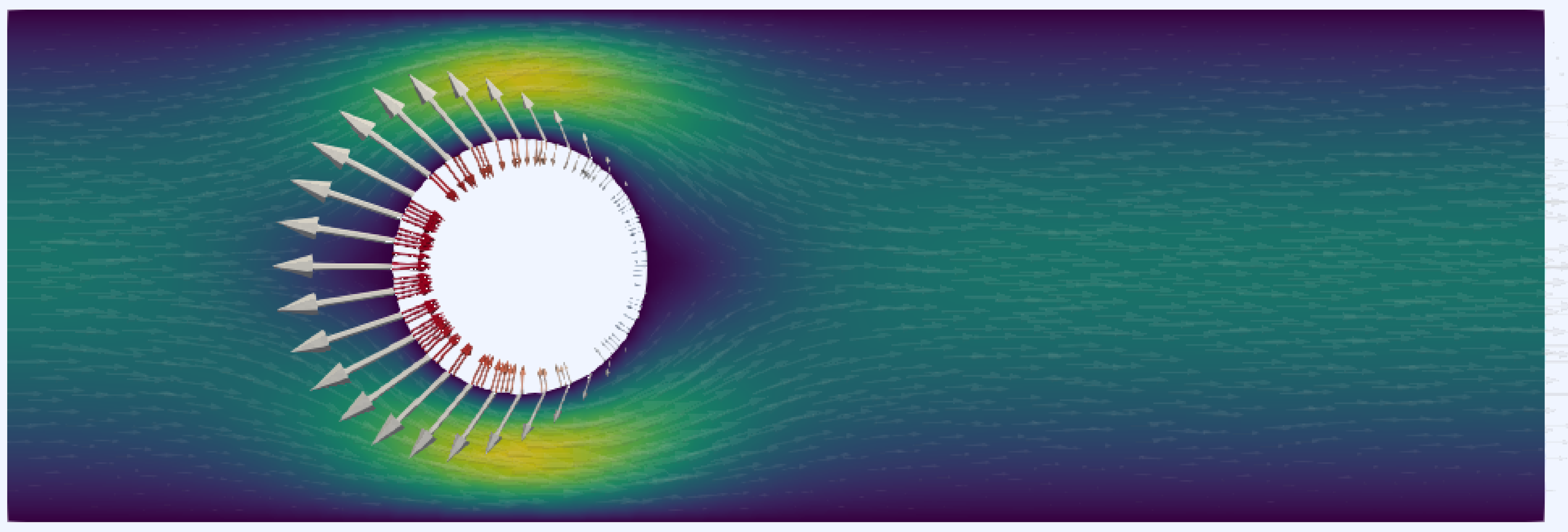
Here Ω is 2D domain, ν is the viscosity, $u : \Omega \to \mathbb{R}^2$ is the unknown velocity, $p : \Omega \to \mathbb{R}$ is the unknown pressure. The goal is to compute the sensitivity of the functional

$$J(u) = \int_\Omega \nabla u \cdot \nabla u \, \mathrm{d}x$$

with respect to the circle boundary function g.

```
from fenics import *
from fenics_adjoint import *

bc = DirichletBC(W.sub(0), g)

a = (nu*inner(grad(u), grad(v))*dx
     - inner(p, div(v))*dx
     - inner(q, div(u))*dx)
L = inner(f, v)*dx

solve(a == L, s, bcs)
u, p = split(s)
J = assemble(inner(u, u)**2*dx)

# Apply fenics-adjoint
dJdg = compute_gradient(J, g)
H = Hessian(J, g)
```

△ **Code**: Simplified FEniCS implementation (complete code has 60 lines)



△ **Figure**: The solution of the code example. The velocity u is shown inside the domain. On the circle boundary, the grey glyphs visualise the l^2 gradient $\mathrm{d}J/\mathrm{d}g$, and the red glyphs visual the Hessian in the negative gradient direction $-H\mathrm{d}J/\mathrm{d}g$.

Time-dependent problem

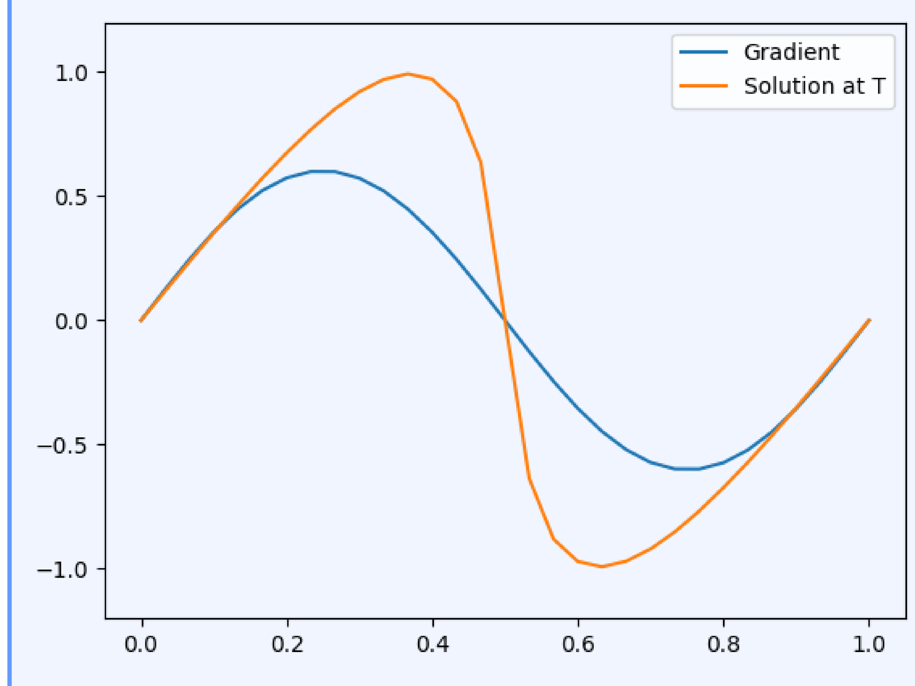
In this example we consider Burger's equation

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

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.

Comparison with dolfin-adjoint

- *90% reduced codebase (lines of code)*;
- *Robust and generic interface to define functionals*;
- *Second-order adjoints*, *Dirichlet boundary controls* and *multiple tapes*.

The AD-tool is still in development. Not all FEniCS features are yet supported. MultiMesh and Firedrake support is planned.

How to get started

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