Differentiable programming accross the PDE/ML divide

David A. Ham[1], Nacime Bouziani[1], India Marsden[2], and Reuben Nixon-Hill[1]

November 2023

[1]Department of Mathematics, Imperial College London [2]Mathematical Institute, University of Oxford

In scientific computing, gradients are *everywhere*

- solving nonlinear PDEs
- computing sensitivities
- data assimilation
- design optimisation
- training neural nets
- ...

Differentiating computer programs

Not a new idea ...

> *LM Beda, LN Korolev, NV Sukkikh, and TS Frolova.* Programs for automatic differentiation for the machine besm.(in russian) technical report, institute for precise mechanics and computation techniques, 1959

by 1981 there are textbooks:

> *Louis B Rall. Automatic differentiation: Techniques and applications.* **Springer, 1981**

So automatic/algorithmic differentiation is nearly as old as computing.

2010s: ML community and ”differentiable programming”.

”Constructing neural networks using pure and higher-order differentiable functions and training them using reverse-mode automatic differentiation is unsurprisingly called Differentiable Programming.”

Erik Meijer. Behind every great deep learning framework is an even greater programming languages concept (keynote).

In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 1, New York, NY, USA, 2018. Association for Computing Machinery

It’s all about the composable abstractions

Abstract Define symbolic representations for numerical objects and algorithms.

Compose Form larger algorithms by plugging together smaller ones.

Claims:

1. There is a useful extension of this concept of differentiable programming to encompass simulation.
2. FEniCS and Firedrake + pyadjoint are examples[1].
3. Using this insight, we can naturally extend packages such as this to interact better with external (non-PDE) processes and data.

[1] Wilkinson Prize 2015. 2011 prize was to Waechter and Laird for IPopt.

So you want to solve a PDE using finite elements

1. Write down a residual, boundary/initial conditions, forcings, parametrisations.
2. Choose suitable finite element paces and quadrature rules.
3. Choose a suitable (non)-linear solver and preconditioning strategy.
4. Derive and implement the loops over elements, facets, basis functions, and quadrature points.
5. Implement parallel communication.
6. Implement and compose solvers and preconditioners.
7. Now do it all again for the adjoint.
8. ...

So you want to solve a PDE using finite elements

1. Write down a residual, boundary/initial conditions, forcings, parametrisations.
2. Choose suitable finite element paces and quadrature rules.
3. Choose a suitable (non)-linear solver and preconditioning strategy.
4. ~~Derive and implement the loops over elements, facets, basis functions, and quadrature points.~~
5. ~~Implement parallel communication.~~
6. ~~Implement and compose solvers and preconditioners.~~
7. ~~Now do it all again for the adjoint.~~
8. ...

About 20 years ago FEniCS worked out how to do this (Kirby & Logg 2006).

10 years later Firedrake joined the party (Rathgeber et al. 2016).

We'll need a PDE then

Burgers Equation:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu\nabla^2 u = 0 \tag{1}$$

$$(n \cdot \nabla)u = 0 \text{ on } \Gamma \tag{2}$$

in weak form: find $u \in V$ such that

$$\int_\Omega \frac{\partial u}{\partial t} \cdot v + ((u \cdot \nabla)u) \cdot v + \nu \nabla u \cdot \nabla v \,\mathrm{d}x = 0 \qquad \forall v \in V_0. \tag{3}$$

For simplicity, use backward Euler in time. At each timestep find $u^{n+1} \in V_0$ such that:

$$\int_\Omega \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla)u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \,\mathrm{d}x = 0 \qquad \forall v \in V_0. \tag{4}$$

```python
from firedrake import *
n = 30
mesh = UnitSquareMesh(n, n)
V = VectorFunctionSpace(mesh, "CG", 2)
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")
v = TestFunction(V)
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)
u_.assign(ic)
u.assign(ic)
nu = 0.0001
timestep = 1.0/n
F = (inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
t = 0.0
end = 0.5
while (t <= end):
    solve(F == 0, u)  # <= all the magic happens here.
    u_.assign(u)
    t += timestep
```

UFL and the FEniCS language were created by the FEniCS project. See Logg et al. 2012
Automated Solution of Differential Equations by the Finite Element Method

Burgers Equation in code

$$\int_\Omega \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla) u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, \mathrm{d}x$$

```
(inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) \
 + nu*inner(grad(u), grad(v)))*dx
```

How does the automation work?

We solve PDEs with Newton-like methods:

$$u_{\mathrm{next}} = u_{\mathrm{cur}} - \left(\frac{\partial F(u_{\mathrm{cur}})}{\partial u}\right)^{-1} F(u_{\mathrm{cur}})$$

So our solver is the composition of a Newton-like algorithm with functions that assemble the residual F and the Jacobian $\partial F/\partial u$.

If V is a real Hilbert space with inner product $\langle\rangle_V$ then V^* is the space of bounded linear functionals $V \to \mathbb{R}$.

The form, given $u \in V$:

$$\int_\Omega u \cdot v \,\mathrm{d}x \quad \forall v \in V \tag{5}$$

is a function $V \to V^*$. This is exactly the form of the residual in a steady PDE.

solve is Differentiable Programming!

This is exactly Meijer's conception of differentiable programming. F is a differentiable operator and the Newton solver is a higher order function.

We can get technical with their signatures:

$$F : V \rightarrow V^* \tag{6}$$

$$\text{Newton} : (V \rightarrow V^*) : V \rightarrow V \tag{7}$$

Imperial College London

UFL does the symbolic maths you would do...

We need to differentiate our residual, F with respect to u. How does a computer do that? Take the nonlinear term from Burgers' equation as an example. You write:

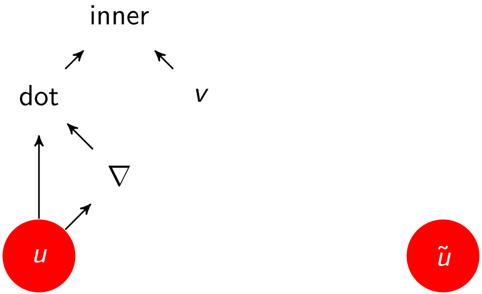
```
inner(dot(u,nabla_grad(u)), v)
```

But the computer sees:

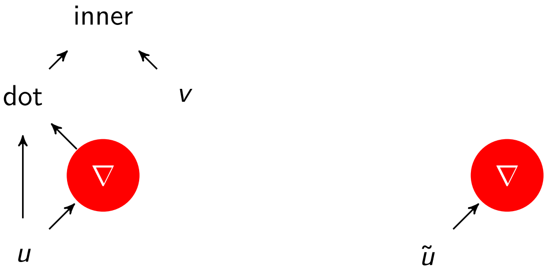
$$
\begin{tikzcd}
 & \mathrm{inner} & \\
\mathrm{dot} \arrow[ur] & & v \arrow[ul] \\
 & \nabla \arrow[ul] & \\
u \arrow[uu] \arrow[ur] & &
\end{tikzcd}
$$

A little symbolic differentiation

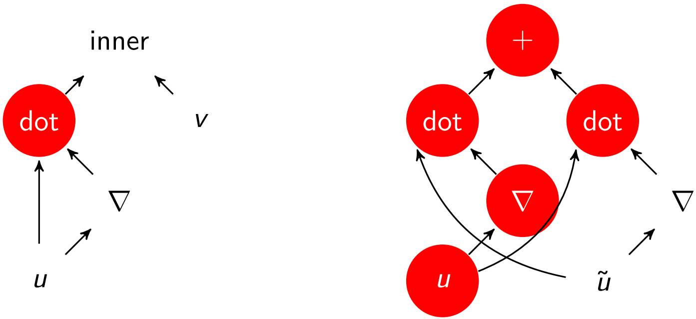
$$\frac{\partial (u \cdot \nabla u) \cdot v}{\partial u} \cdot \tilde{u} = ?$$

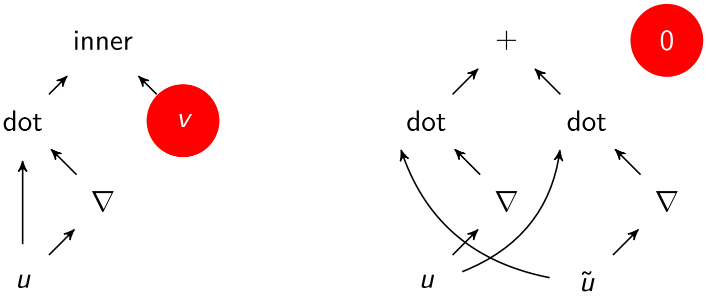


A little symbolic differentiation

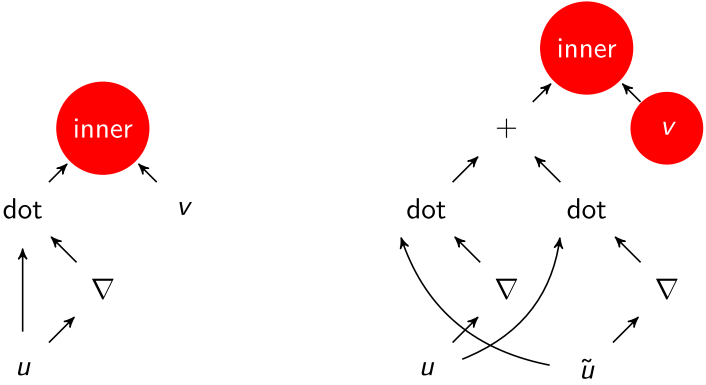


A little symbolic differentiation




inner
dot
v
∇
u
+
0
dot
dot
∇
∇
u
ũ

A little symbolic differentiation



$$\frac{\partial (u \cdot \nabla u) \cdot v}{\partial u} \cdot \tilde{u} = (\tilde{u} \cdot \nabla u + u \cdot \nabla \tilde{u}) \cdot v$$

Adjoint PDEs

Turns out scientists and engineers usually solve inverse problems:

- Sensitivity analysis,
- Parameter estimation,
- Design optimisation,
- Data assimilation.

Common to all of these is a requirement to differentiate the model.

This work rests on Pyadjoint by Mitusch, Funke and Dokken, and Dolfin-adjoint by Farrell, Funke, Ham and Rognes.

At each timestep find $u^{n+1} \in V_0$ such that:

$$\int_\Omega \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla)u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, \mathrm{d}x = 0 \qquad \forall v \in V_0. \tag{8}$$

Let's write a simple functional:

$$J(u) = \int_\Omega u^2_{t=\text{end}} + u^2_{t=0} \mathrm{d}x$$

and assume that we want to differentiate this with respect to the initial condition $u_{t=0}$.

What would that do to our code?

Types of derivative

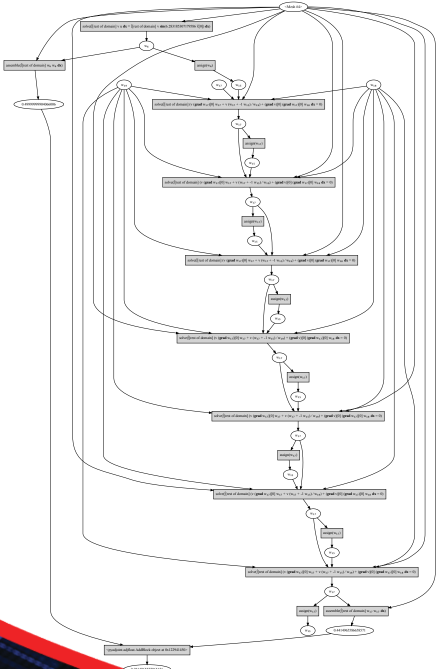
Symbolic differentiation Derivative of all outputs with respect to all inputs at an arbitrary state.

Forward mode/tangent linear model Derivative of all outputs with respect to one input at a single given state.

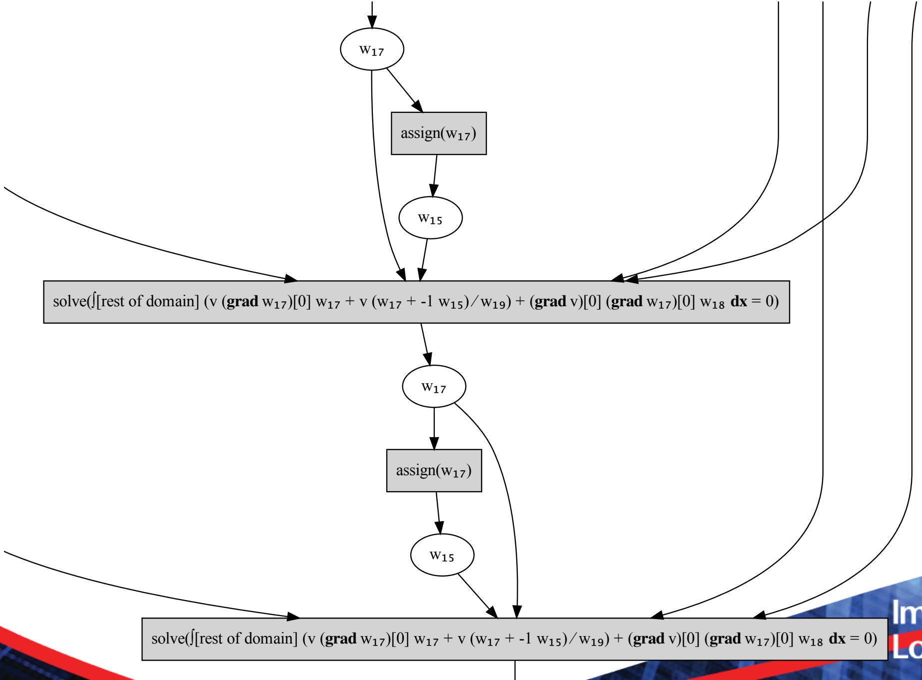
Reverse mode/adjoint/backpropagation Derivative of a single output with respect to all inputs at a single given state.

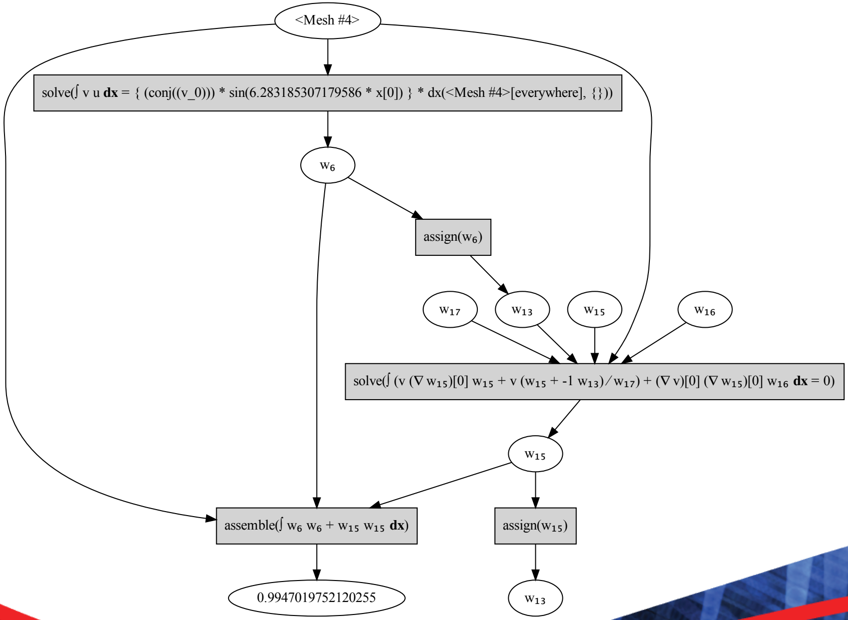
Differentiation with pyadjoint

```
from firedrake import *
from firedrake.adjoint import *
continue_annotation()
n = 30
mesh = UnitSquareMesh(n, n)
V = VectorFunctionSpace(mesh, "CG", 2)
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")
v = TestFunction(V)
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)
u_.assign(ic)
u.assign(ic)
nu = 0.0001
timestep = 1.0/n
F = (inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
t = 0.0
end = 1.0
while (t <= end):
    solve(F == 0, u)
    u_.assign(u)
    t += timestep
J = assemble(u*u*dx + ic*ic*dx)
compute_gradient(J, Control(ic))
```



Imperial College London


w17
assign(w17)
w15
solve(∫[rest of domain] (v (grad w17)[0] w17 + v (w17 + -1 w15)/w19) + (grad v)[0] (grad w17)[0] w18 dx = 0)
w17
assign(w17)
w15
solve(∫[rest of domain] (v (grad w17)[0] w17 + v (w17 + -1 w15)/w19) + (grad v)[0] (grad w17)[0] w18 dx = 0)



Imperial College London

An adjoint block

For each operation:

$$y = f(x) \tag{9}$$

We compute the adjoint (transpose derivative):

$$x' = \left(\frac{\partial f}{\partial x}(x)\right)^* y' \tag{10}$$

Solve:

$$F(u; v) = 0 \tag{11}$$

the implicit function theorem gives us:

$$u' = \left(\frac{\partial F}{\partial u}(u; \tilde{u}, v)\right)^{-*} \lambda \tag{12}$$

$$= \left(\frac{\partial F}{\partial u}(u; v, \tilde{u})\right)^{-1} \lambda \tag{13}$$

Newton as a differentiable operator

Back to the function signatures, expanded for unsteady:

$$F : \underbrace{V}_{u_{\mathrm{old}}} \times \underbrace{V}_{u_{\mathrm{new}}} \to V^* \tag{14}$$

$$\text{Newton} : (V \times V \to V^*) : \underbrace{V}_{u_{\mathrm{old}}} \to \underbrace{V}_{u_{\mathrm{new}}} \tag{15}$$

$$\text{Newton}^* : (V \to V^*) : \underbrace{V}_{u_{\mathrm{old}}} \times \underbrace{V}_{u_{\mathrm{new}}} \times \underbrace{V^*}_{u'_{\mathrm{new}}} \to \underbrace{V^*}_{u'_{\mathrm{old}}} \tag{16}$$

Imperial College London

This tells us how to expand beyond just PDEs.

Suppose what I need is find $u \in V$ such that:

$$F(u, N(u); v) = 0 \quad \forall v \in V^* \tag{17}$$

Where F is a PDE residual but N is not. N could be a parametrisation whose value is give by e.g.:

1. Solving an ODE at each point.
2. Solving an algebraic equation at each point.
3. Evaluating a neural net.

The same applies to inverse problems

$$\min_{u \in V} \|u - u_{\mathrm{obs}}\| + N(u) \tag{18}$$

Subject to:

$$F(u; v) = 0 \quad \forall v \in V \tag{19}$$

Where N is a regularisation term e.g. found by evaluating a neural net.

Imperial College London

We just need composable differentiable operators

UFL External operators:

1. Symbolic behaviour (given by calculus rules).
2. Numerical implementation (not UFL's problem).

Just a few more arrows

UFL is the Unified **Form** Language:

$$V \times W \to \mathbb{R} \tag{20}$$

but the external operators we're interested in aren't (obviously) forms:

$$V \to W \tag{21}$$

A really simple idea: we can apply function arguments one at a time:

$$V \times W \to \mathbb{R} \equiv V \to (W \to \mathbb{R}) \tag{22}$$

A really simple idea: we can apply function arguments one at a time:

$$V \times W \to \mathbb{R} \equiv V \to (W \to \mathbb{R}) \tag{22}$$

$$\equiv V \to W^* \tag{23}$$

So any form is an operator into the dual space of its last argument.

But we need the other direction

Happily, all the spaces we care about are reflexive:

$$V \Leftrightarrow V^{**} \; (\equiv V^{*} \to \mathbb{R}) \tag{24}$$

by identifying $v^{**} \in V^{**}$ with $v \in V$ such that:

$$v^{**}(u^{*}) = u^{*}(v) \quad \forall u^{*} \in V^{*} \tag{25}$$

Imperial College London

But we need the other direction

Happily, all the spaces we care about are reflexive:

$$V \Leftrightarrow V^{**} \; (\equiv V^* \to \mathbb{R}) \tag{24}$$

by identifying $v^{**} \in V^{**}$ with $v \in V$ such that:

$$v^{**}(u^*) = u^*(v) \quad \forall u^* \in V^* \tag{25}$$

Hence:

$$\begin{aligned} V \to W &\equiv V \to W^{**} && (26)\\ &\equiv V \to (W^* \to \mathbb{R}) && (27)\\ &\equiv V \times W^* \to \mathbb{R} && (28) \end{aligned}$$

Imperial College London

UFL external operator

```
X = FunctionSpace(...)
V = FunctionSpace(...)
u = Coefficient(V)
m = Coefficient(V)
v = TestFunction(V)
uhat = TrialFunction(V)

# N : V x V x V* -> ℝ
#       u, m, v*   -> N(u, m; v*)
N = ExternalOperator(u, m, function_space=X)

# Define a given form F
F = u * N * v * dx

# Symbolically compute the derivative ∂N(u,m;û,v*)/∂u
dNdu = derivative(N, u, uhat)

# Symbolically compute the derivative dF/du
dFdu = derivative(F, u, uhat)
```


Imperial College London

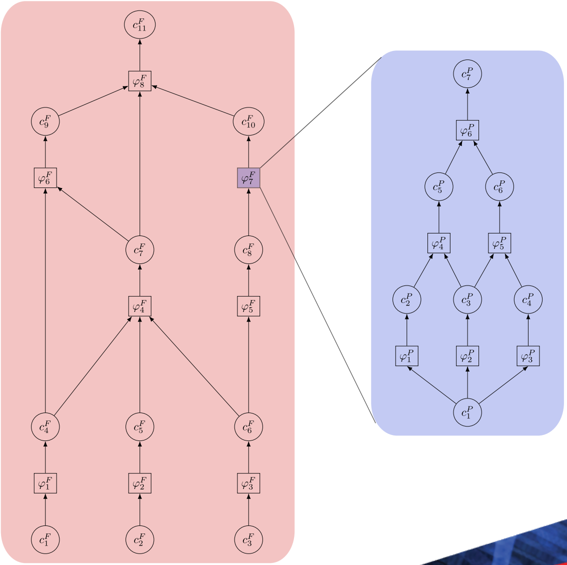
External operators obey the right symbolic rules.

UFL expression	ExternalOperator	Derivatives	Argument slots	Output type
N	$N(u, m; v^*)$	$(0, 0)$	$(v^*,)$	Coefficient
$dNdu = \mathrm{derivative}(N, u, \hat{u})$	$\frac{\partial N(u,m;\hat{u},v^*)}{\partial u}$	$(1, 0)$	$(v^*, \hat{u})$	Matrix
$dNdm = \mathrm{derivative}(N, m, \hat{m})$	$\frac{\partial N(u,m;\hat{m},v^*)}{\partial m}$	$(0, 1)$	$(v^*, \hat{m})$	Matrix
action(dNdu, w)	$\frac{\partial N(u,m;w,v^*)}{\partial u}$	$(1, 0)$	(v^*, w)	Coefficient
adjoint(dNdm)	$\frac{\partial N(u,m;v^*,\hat{m})}{\partial m}$	$(0, 1)$	$(\hat{m}, v^*)$	Matrix
action(adjoint(dNdm), ṽ)	$\frac{\partial N(u,m;\tilde{v},\hat{m})}{\partial m}$	$(0, 1)$	$(\hat{m}, \tilde{v})$	Coefficient

The user provides the implementation

```
class MyExternalOperator(AbstractExternalOperator):
  def __init__(self, *args, **kwargs):
    ...
  @assemble_method((0, 0), (0,))
  # or @assemble_method(0, (0,))
  def N(self, *args, *kwargs):
    """Evaluate my external operator N"""
    ...
  @assemble_method((1, 0), (0, 1))
  def dNdu(self, *args, **kwargs):
    """Evaluate dNdu"""
    ...
  @assemble_method((1, 0), (0, None))
  def dNdu_action(self, *args, **kwargs):
    """Evaluate the action of dNdu"""
    ...
  @assemble_method((0, 1), (1, 0))
  def dNdm_adjoint(self, *args, **kwargs):
    """Evaluate dNdm*"""
    ...
  @assemble_method((0, 1), (None, 0))
  def dNdm_adjoint_action(self, *args, **kwargs):
    """Evaluate the action of dNdm*"""
    ...
```

Suppose we want to define a neural net operator.



Suppose we want to define a neural net operator.

PyTorch	Firedrake
$f(x^P;\theta)$	$N(x^F; v^*)$
$\text{jacobian}(f, x^P)$	$J = \text{derivative}(N, x^F)$ $\text{assemble}(J)$
$\text{jvp}(f, x^P, z^P)$	$\text{assemble}(\text{action}(J, z^F))$
$\text{vjp}(f, x^P, z^P)$	$\text{assemble}(\text{action}(\text{adjoint}(J), z^F))$
$\text{hvp}(f, x^P, z^P)$	$H = \text{derivative}(J, x^F)$ $\text{assemble}(\text{action}(H, z^F))$
$\text{vhp}(f, x^P, z^P)$	$\text{assemble}(\text{action}(\text{adjoint}(H), z^F))$

Turns out PyTorch has all of our operators.

PyTorch operator

```
import torch
from torch.autograd.functional import jvp
...
class PytorchOperator(MLOperator):
    ...
    @assemble_method(0, (0,))
    def forward(self, *args, **kwargs):
        V = self.function_space()
        x^F, _ = self.ufl_operands
        # Convert input to PyTorch
        x^P = self.ml_backend.to_ml_backend(x^F)
        # Forward pass
        y^P = self.model(x^P)
        # Convert output to Firedrake
        y^F = self.ml_backend.from_ml_backend(y^P, V)
        return y^F
    ...
```


Imperial College London

A simple tomography example

$$\min_{c \in P} \quad \frac{1}{2}\|\varphi - \varphi^{obs}\|_V + \alpha\mathcal{R}(c) \tag{29}$$

subject to:

$$F(\varphi, c; v) = 0 \quad \forall v \in V \tag{30}$$

Tomography code

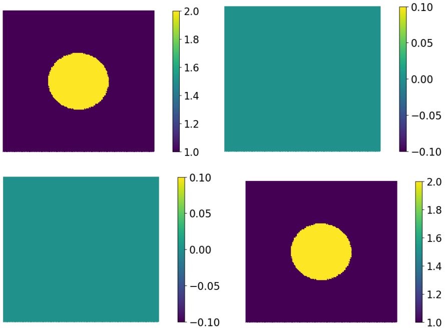
```
from firedrake import *
from firedrake_adjoint import *

...
# Get a pre-trained PyTorch model
model = ...
# Define the external operator from the model
pytorch_op = neuralnet(model, function_space=...)
N = pytorch_op(vel)

# Solve the forward problem defined by equation (??)
solve(F(c, phi, v) == 0, phi, ...)
# Assemble the cost function:
J = assemble(0.5*(inner(phi-phi_obs, phi-phi_obs) +
                  alpha*inner(N, N))*dx)

# Optimise the problem
Jhat = ReducedFunctional(J, Control(c))
c_opt = minimize(Jhat, method="L-BFGS-B", tol=1.0e-7,
                  options={"disp": True, "maxiter" : 20})
```

The only computational result in this talk



Recovered wave speed c as a function of position (x, z): exact velocity (upper left), without regularisation (upper right), Tikhonov regulariser (lower left), neural network-based regulariser (lower right).

Other composable abstraction layers in and around Firedrake:

Fireshape Shape optimisation (Alberto Paganini, Leicester)

Deflated continuation Finding multiple solutions to nonlinear PDEs (Patrick Farrell, Oxford)

Point data operators Interact with real data and have first class point sources (Reuben Nixon-Hill)

Firedrake

UK Research and Innovation

Natural Environment Research Council

Engineering and Physical Sciences Research Council