Firedrake

Automating Finite Element Simulation by Generating Tensor Computations from Vector Calculus

David A. Ham[1] and the Firedrake team

January 2022

[1]Department of Mathematics, Imperial College London

Imperial College London

Current Firedrake Team

- Lawrence Mitchell, Durham: Solvers, preconditioners, funny elements, data structures, the kitchen sink ...
- Koki Sagiyama, Imperial: Multidomain, coupling and I/O
- Jack Betteridge, Imperial: Everything HPC
- Nacime Bouziani, Imperial: External operators
- Sophia Vorderwuelbecke, Imperial: High order methods, SLATE, vectorisation
- Reuben Nixon-Hill, Imperial: Interpolation and data assimilation
- Connor Ward, Imperial: Code generation infrastructure and performance
- Robert Kirby, Baylor: Weird elements

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. ...

You specify the maths, and Firedrake does the rest. But how?

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:

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

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.

Burgers Equation in code

```
1  from firedrake import *
2  n = 30
3  mesh = UnitSquareMesh(n, n)
4  V = VectorFunctionSpace(mesh, "CG", 2)
5  u_ = Function(V, name="Velocity")
6  u = Function(V, name="VelocityNext")
7  v = TestFunction(V)
8  x = SpatialCoordinate(mesh)
9  ic = project(as_vector([sin(pi*x[0]), 0]), V)
10 u_.assign(ic)
11 u.assign(ic)
12 nu = 0.0001
13 timestep = 1.0/n
14 F = (inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
15 t = 0.0
16 end = 0.5
17 while (t <= end):
18     solve(F == 0, u)  # <= all the magic happens here.
19     u_.assign(u)
20     t += timestep
```

How does the automation work?

We solve PDEs with Newton-like methods:

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

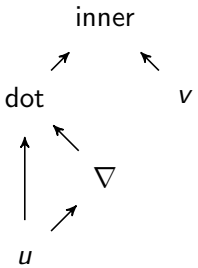
So our solver is the composition of a Newton-like algorithm with functions that assemble the residual F and the Jacobian $\partial F / \partial u$.

Firedrake 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:

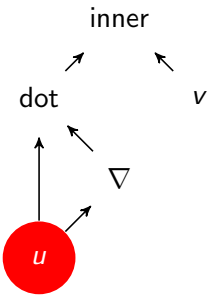
```
inner(dot(u,nabla_grad(u)), v)
```

But the computer sees:

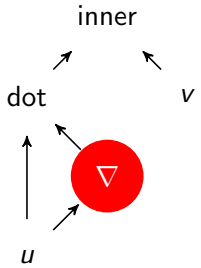


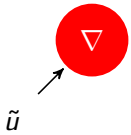
A little algorithmic differentiation

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

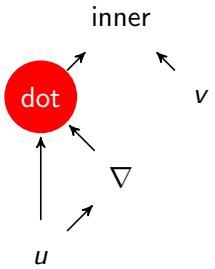


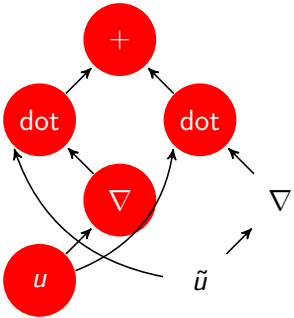
A little algorithmic differentiation



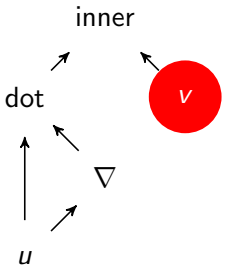


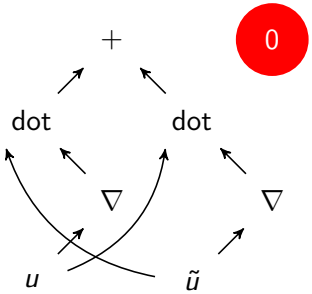
A little algorithmic differentiation



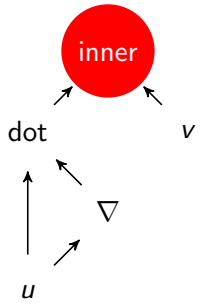


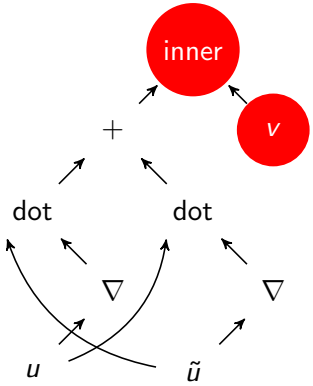
A little algorithmic differentiation





A little algorithmic differentiation





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

We now have $F(u)$ and $\partial F(u)/\partial u$ as symbolic objects, but we need to evaluate those integrals.

Same principles: visit the expression tree node by node.

Evaluate integrals element-wise:

$$\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x = \sum_c \int_c \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x$$

Evaluate integrals element-wise:

$$\int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x = \sum_c \int_c \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x$$

Transform to the reference cell:

$$\int_c \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x = \int_{c_0} J^{-T}\nabla\Phi_{\hat{i}} \cdot J^{-T}\nabla\Phi_{\hat{j}} |J| \mathrm{d}X$$

where capital letters indicate quantities in reference cell coordinates.

Then we replace the integrals with suitable quadrature:

$$\sum_q J_q^{-T} \nabla \Phi_{\hat{i}}(X_q) \cdot J_q^{-T} \nabla \Phi_{\hat{j}}(X_q) |J_q| w_q$$

with:

$$J_q = \sum_{\hat{k}} x_{\hat{k}} \nabla \Psi_{\hat{k}}(X_q)$$

where $x_{\hat{k}}$ are the nodal values of the coordinate field and $\Psi_{\hat{k}}$ is the local basis for the coordinate space.

Operator evaluation

Evaluating integrals by quadrature requires tabulations of the finite element bases, and their derivatives. Happily FIAT (Kirby, 2004) + FInAT provides exactly this functionality for a huge range of elements.

The local operation therefore reduces to a tensor contraction:

$$\int_c \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x = \sum_{\alpha\,\beta\,\gamma\,q} \left(J_q^{-1}\right)_{\beta\,\alpha} P_{\hat{i}\,\beta\,q} \left(J_q^{-1}\right)_{\gamma\,\alpha} P_{\hat{j}\,\gamma\,q} |J_q| w_q$$

with:

$$J_{q\,\alpha\,\beta} = \sum_{\hat{k}} x_{\hat{k}} Q_{\hat{k}\,\alpha\,\beta\,q}$$

and tabulation matrices:

$$P_{\hat{i}\,\alpha\,q} = \frac{\partial \Phi_{\hat{i}}(X_q)}{\partial X_\alpha} \qquad Q_{\hat{i}\,\alpha\,\beta\,q} = \frac{\partial \Psi_{\alpha\hat{i}}(X_q)}{\partial X_\beta}$$

So how do we do that fast?

$$\int_c \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}x = \sum_{\alpha\,\beta\,\gamma\,q} \left(J_q^{-1}\right)_{\beta\,\alpha} P_{\hat{i}\,\beta\,q} \left(J_q^{-1}\right)_{\gamma\,\alpha} P_{\hat{j}\,\gamma\,q} |J_q| w_q$$

The order in which this sum occurs radically affects the number of operations and size of temporaries: which is a constrained ILP!

Suppose now I do this on hexahedral elements:

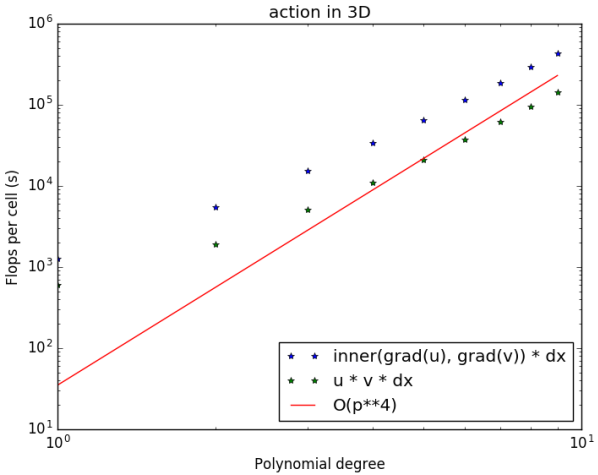
$$\int_c \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}x = \sum_{\alpha\,\beta\,\gamma\,\hat{i}_0\,\hat{i}_1,\hat{i}_2,\hat{j}_0\,\hat{j}_1\,\hat{j}_2\,q_0\,q_1\,q_2} \left(J_q^{-1}\right)_{\beta\,\alpha} \begin{bmatrix} \mathrm{d}P_{\hat{i}_0\,q_0} P_{\hat{i}_1\,q_1} P_{\hat{i}_2\,q_2} \\ P_{\hat{i}_0\,q_0} \mathrm{d}P_{\hat{i}_1\,q_1} P_{\hat{i}_2\,q_2} \\ P_{\hat{i}_1\,q_0} P_{\hat{i}_1\,q_1} \mathrm{d}P_{\hat{i}_2\,q_2} \end{bmatrix}_\beta \left(J_q^{-1}\right)_{\gamma\,\alpha} \begin{bmatrix} \mathrm{d}P_{\hat{j}_0\,q_0} P_{\hat{j}_1\,q_1} P_{\hat{j}_2\,q_2} \\ P_{\hat{j}_0\,q_0} \mathrm{d}P_{\hat{j}_1\,q_1} P_{\hat{j}_2\,q_2} \\ P_{\hat{j}_1\,q_0} P_{\hat{j}_1\,q_1} \mathrm{d}P_{\hat{j}_2\,q_2} \end{bmatrix}_\gamma |J_q| w_{q_0} w_{q_1} w_{q_2} \quad (5)$$

FIAT can give us P or $\mathrm{d}P$, and FInAT can give us the expressions in those terms which we then factorise.

The naïve implementation is $O(p^9)$!

If you do everything right (including in the solvers) it's $O(p^4)$.

The proof of the pudding:



Our nonlinear PDE looks like:

$$F(u; v) = 0 \qquad \forall v \in V$$

Which we solve with a Newton-like iteration over linear solves:

$$N(F, J, K)$$

Where $F(u)$ is the function which assembles the residual, $J(u)$ is the function which assembles $\partial F(u)/\partial u$ and K is a linear solver.

Work by Thomas Gibson, now being taken forward by Sophia Vorderwuelbecke and all building on the PETSc composable solver framework.

PETSc represents linear solvers as a preconditioned Krylov subspace method:

$$P(K(J, F))$$

Where the preconditioner P takes $K(J, F)$ to another (hopefully more tractable) solver. A classical left preconditioner is given by:

$$P_l(\hat{K}, A)(K(J, F)) = K(\hat{K}(A, J), \hat{K}(A, F))$$

Where A is another matrix operator, and $\hat{K}$ is another Krylov subspace method. Now it's preconditioners all the way down!

Linear rotating shallow water equations:

$$\boldsymbol{u_t} + \boldsymbol{f u^\perp} + \boldsymbol{g \nabla D} = 0,$$
$$\boldsymbol{D_t} + \boldsymbol{H \nabla \cdot u} = 0$$

Discretizing in time and space yields the indefinite linear operator:

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{pmatrix}$$

Compatible FE formulation:
Find $(u, D) \in \mathbb{V}_1 \times \mathbb{V}_2$ such that

$$\langle \boldsymbol{w}, \boldsymbol{u_t} \rangle_{\Omega_h} + \boldsymbol{f} \left\langle \boldsymbol{w}, \boldsymbol{u^\perp} \right\rangle_{\Omega_h} + \boldsymbol{g} \langle \boldsymbol{\nabla \cdot w}, \boldsymbol{D} \rangle_{\Omega_h} = 0,$$
$$\langle \boldsymbol{\phi}, \boldsymbol{H \nabla \cdot u} \rangle_{\Omega_h} + \langle \boldsymbol{\phi}, \boldsymbol{D_t} \rangle_{\Omega_h} = 0$$

for all $(w, \phi) \in \mathbb{V}_1 \times \mathbb{V}_2$.

Schur complement approach
Schur complement precondtioners seek to approximate

$$\mathcal{P} \approx \begin{pmatrix} \boldsymbol{I} & -\boldsymbol{A}^{-1}\boldsymbol{B} \\ 0 & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{A}^{-1} & 0 \\ 0 & \boldsymbol{S}^{-1} \end{pmatrix} \begin{pmatrix} \boldsymbol{I} & 0 \\ -\boldsymbol{C}\boldsymbol{A}^{-1} & \boldsymbol{I} \end{pmatrix}$$

where $S = D - CA^{-1}B$ is the Schur complement of the orignal operator with respect to A.

The main source of trouble: A^{-1}

(since functions in $\mathbb{V}_1$ have continuous normals across boundaries).

Hybrid-Mixed methods (Arnold & Brezzi 1985)

We can reformulate the problem in terms of globally discontinuous functions.

- $\widetilde{\mathbb{V}}_1$ is the space of velocities with discontinuous normals.
- $\mathbb{T}(\mathbb{V}_1) = \mathbb{T}$ is the space of approximate traces.

Now we discretise in time and space:

$$\langle \boldsymbol{w}, \tilde{\boldsymbol{u}}_t \rangle_{\Omega_h} + \boldsymbol{f} \left\langle \boldsymbol{w}, \tilde{\boldsymbol{u}}^{\perp} \right\rangle_{\Omega_h} + \boldsymbol{g} \langle \boldsymbol{\nabla} \cdot \boldsymbol{w}, \boldsymbol{D} \rangle_{\Omega_h} - \sum_{K \in \Omega_h} \langle\!\langle \boldsymbol{w} \cdot \boldsymbol{n}, \boldsymbol{\lambda} \rangle\!\rangle_{\partial K \setminus \partial\Omega} = 0$$

$$\langle \phi, \boldsymbol{H} \boldsymbol{\nabla} \cdot \tilde{\boldsymbol{u}} \rangle_{\Omega_h} + \langle \phi, \boldsymbol{D}_t \rangle_{\Omega_h} = 0$$

$$\sum_{K \in \Omega_h} \langle\!\langle \gamma, \tilde{\boldsymbol{u}} \cdot \boldsymbol{n} \rangle\!\rangle_{\partial K \setminus \partial\Omega} = 0$$

for all $(w, \phi, \gamma) \in \widetilde{\mathbb{V}}_1 \times \mathbb{V}_2 \times \mathbb{T}$.

The global system for the hybridised equations:

$$\begin{pmatrix} \widetilde{A} & B & K^T \\ C & D & 0 \\ K & 0 & 0 \end{pmatrix} \begin{Bmatrix} \tilde{u} \\ D \\ \lambda \end{Bmatrix} = \begin{Bmatrix} \mathcal{R}_w \\ \mathcal{R}_\phi \\ 0 \end{Bmatrix}.$$

We can directly compute the Schur complement system in a cell-local manner:

$$\begin{pmatrix} K & 0 \end{pmatrix} \begin{pmatrix} \widetilde{A} & B \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} K^T \\ 0 \end{pmatrix} \lambda = \begin{pmatrix} K & 0 \end{pmatrix} \begin{pmatrix} \widetilde{A} & B \\ C & D \end{pmatrix}^{-1} \begin{Bmatrix} \mathcal{R}_w \\ \mathcal{R}_\phi \end{Bmatrix}$$

$\tilde{u}$ and D can be reconstructed from λ element-wise.

The Slate Language

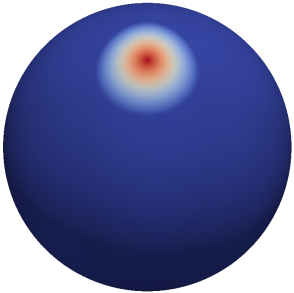
Slate is a DSL for expressing localised linear algebra on finite element tensors. Each elemental tensor is associated with a UFL form.

```
[...] # Define test and trial functions £w, u, \phi, D£
# Define the Slate tensor corresponding to the mixed ("broken") operator
M = Tensor((dot(w, u) + div(w)*D + phi*D + phi*div(u)) * dx)
# Lagrange multipliers on interior facets (test functions £\gamma£)
K = Tensor(gammar('+')*dot(u, n) * dS)
S = assemble(K * M.inv * K.T, bcs=[...])
```

Hybridization as a preconditioner

- PETSc already provides a highly runtime-configurable library for algebraically composing solvers (Brown et al. 2012).
- Firedrake makes it straightforward to build auxiliary operators (Lawrence Mitchell & Rob Kirby).
- Slate provides a linear algebra context for local operators.
- Combining these, we can automate the hybridization process in the form of a custom python preconditioner: `firedrake.HybridizationPC`

Williamson mountain test case



Topography for the Williamson mountain test case
(Peak approx. 2000m)

```
class ShallowWaterSolver(TimesteppingSolver):

  def _setup_solver(self):
      [...]  # Set up constants/coefficients and parameters
      w, phi = TestFunctions(W)
      u, D = TrialFunctions(W)
      eqn = (inner(w, u) - beta*g*div(w)*D - inner(w, u_in)
             + phi*D + beta*H*phi*div(u) - phi*D_in) * dx
      a = lhs(eqn)
      L = rhs(eqn)
      # Set up the variational problem and a hybridised solver
      x = self.state.dy
      problem = LinearVariationalProblem(a, L, x)
      params = {'ksp_type': 'preonly',
                'mat_type': 'matfree',
                'pc_type': 'python',
                'pc_python_type': 'firedrake.HybridizationPC',
                'hybridization': {'ksp_type': 'cg',
                                  'pc_type': 'gamg'}}
      solver = LinearVariationalSolver(problem, params)
      self.solver = solver

  def solve(self):
      self.solver.solve()
```

2M DoFs on 32 cores.

Stage	**Hybridization**		**Schur Comp**	
	Avg Time (s)	% Total	Avg Time (s)	% Total
Apply forcing:	1.9009e+02	17.3%	1.9175e+02	4.2%
Advection:	3.6147e+02	32.8%	3.6120e+02	7.9%
Implicit solve:	4.6381e+02	42.1%	3.9335e+03	85.9%

Implicit solve	**Hybridization** Iterations	**Schur Comp** Iterations
Outer solve (GMRES)	0	11
Inner solve (CG)	4	5

Table 1: PETSc performance summary for a 15min computational time run (above) and iterations to reach convergence for implicit solve (below).

Other composable abstraction layers in and around Firedrake:

Adjoint Automated inverse problems.

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

External operator interface Plug in neural nets and other non-PDE operators (Nacime Bouziani)

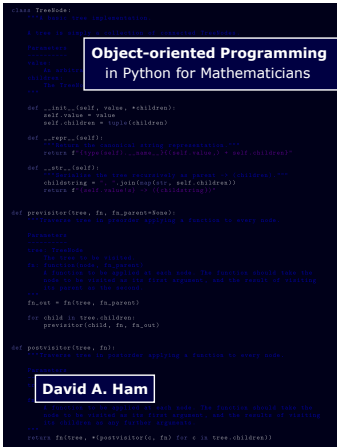
Point data operators Interact with real data (Reuben Nixon-Hill)

Some of the Firedrake applications

- Quasigeostrophic turbulence (Waterloo)
- Numerical schemes for atmospheric flow (Imperial, Exeter, Met Office)
- Improving prediction of fronts (Imperial)
- Estuarine and coastal flows (Finnish Met. Institute)
- Optimal location of marine power resources (Imperial)
- Fluid structure interaction (Leeds)
- Multiphase flow in porous media (Aachen)
- Shape optimisation (Leicester)
- Liquid crystal structure (Oxford, Memorial University)
- Ice sheet and glacier flows (University of Washington)
- Seismic imaging (University of São Paulo)
- Earth mantle dynamics (Australian National University)

Known users on 6 continents.

Shameless plug. . .



Out now.

Firedrake is likely to be hiring one or more postdocs shortly. Please talk to me if interested.

Firedrake

UK Research and Innovation

Natural Environment Research Council

Engineering and Physical Sciences Research Council