Firedrake: the architecture of a compiler that automates the finite element method

Paul Kelly
Group Leader, Software Performance Optimisation
Department of Computing
Imperial College London

Joint work with David Ham (Imperial Maths), Lawrence Mitchell (Imperial Computing) Fabio Luporini (Imperial Earth Science Engineering), Florian Rathgeber (now with Google), Doru Bercea (now with IBM Research), Michael Lange (now with ECMWF), Andrew McRae (now at University of Oxford), Graham Markall (now at Embecosm Ltd), Tianjiao Sun (now at Cerebras), Thomas Gibson (Imperial Maths) And many others....
This talk

- Three different potential audiences:
  - Programming language design and implementation
  - Numerical methods for PDEs
  - High-performance computing

- What is Firedrake?

- What is it used for? By whom?

- What does its DSL actually look like?

- What is its domain of applicability?

- How is its compiler designed?

- Does it generate good code?

- Does it automate interesting optimisations that would be hard to do by hand?

- What are the open research challenges?

- What would we do differently?

- What is the opportunity to change the world?
Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM). Firedrake uses sophisticated code generation to provide mathematicians, scientists, and engineers with a very high productivity way to create sophisticated high performance simulations.

Features:

- Expressive specification of any PDE using the Unified Form Language from the FEniCS Project.
- Sophisticated, programmable solvers through seamless coupling with PETSc.
- Triangular, quadrilateral, and tetrahedral unstructured meshes.
- Layered meshes of triangular wedges or hexahedra.
- Vast range of finite element spaces.
- Sophisticated automatic optimisation, including sum factorisation for high order elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.
Firedrake is an automated system for the solution of partial differential equations (FEM). Firedrake uses sophisticated code generation tools to enable mathematicians, scientists, and engineers with a very high productivity way of building and solving complex computational science problems.

Features:

- Expressive specification of any PDE using the Unified Form Language (UFL).
- Sophisticated, programmable solvers through seamless coupling with libraries such as PETSc and Trilinos.
- Triangular, quadrilateral, and tetrahedral unstructured meshes.
- Layered meshes of triangular wedges or hexahedra.
- Vast range of finite element spaces.
- Sophisticated automatic optimisation, including sum factorisation for matrix-free elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.

**What is Firedrake?**
Firedrake is used in:

**Thetis**: unstructured grid coastal modelling framework

---

The Thetis project

Thetis is an unstructured grid coastal ocean model built using the Firedrake finite element framework. Currently Thetis consists of 2D depth averaged and full 3D baroclinic models. Some example animations are shown below. More animations can be found in the Youtube channel.

---

What is it used for? By whom?

---

Current development status

Latest status: build passing

Thetis source code is hosted on Github and is being continually tested using Jenkins.
Tidal barrage simulation using Thetis ([https://thetisproject.org/](https://thetisproject.org/))

What is it used for? By whom?
Firedrake is used in:

- **Gusto**: atmospheric modelling framework being used to prototype the next generation of weather and climate simulations for the UK Met Office.


What is it used for? By whom?
Firedrake is used in:

**Icepack**: a framework for modeling the flow of glaciers and ice sheets, developed at the Polar Science Center at the University of Washington.

Larsen ice shelf model, from the Icepack tutorial by Daniel Shapero ([https://icepack.github.io/icepack.demo.02-larsen-ice-shelf.html](https://icepack.github.io/icepack.demo.02-larsen-ice-shelf.html))
The finite element method in outline

\[ \text{do element} = 1, N \]
\[ \text{assemble}(\text{element}) : \]
\[ \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX. \]
\[ \text{end do} \]

\[ Ax = b \]

Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector.
Multilayered abstractions for FE

Local assembly:
- Computes local assembly matrix
- Using:
  - The (weak form of the) PDE
  - The discretisation
- Key operation is evaluation of expressions over basis function representation of the element

Mesh traversal:
- PyOP2
- Loops over the mesh
- Key is orchestration of data movement

Solver:
- Interfaces to standard solvers through PetSc
Example: Burgers equation

We start with the PDE: (see https://www.firedrakeproject.org/demos/burgers.py.html)

The Burgers equation is a non-linear equation for the advection and diffusion of momentum. Here we choose to write the Burgers equation in two dimensions to demonstrate the use of vector function spaces:

\[
\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu \nabla^2 u = 0
\]

\[(n \cdot \nabla)u = 0 \text{ on } \Gamma\]

where $\Gamma$ is the domain boundary and $\nu$ is a constant scalar viscosity. The solution $u$ is sought in some suitable vector-valued function space $V$. We take the inner product with an arbitrary test function $v \in V$ and integrate the viscosity term by parts:

\[
\int_{\Omega} \frac{\partial u}{\partial t} \cdot v + ((u \cdot \nabla)u) \cdot v + \nu \nabla u \cdot \nabla v \, dx = 0.
\]

The boundary condition has been used to discard the surface integral. Next, we need to discretise in time. For simplicity and stability we elect to use a backward Euler discretisation:

\[
\int_{\Omega} \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla)u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, dx = 0.
\]

From the weak form of the PDE, we derive an equation to solve, that determines the state at each timestep in terms of the previous timestep.
From the weak form of the PDE, we derive an equation to solve, that determines the state at each timestep in terms of the previous timestep.
Example: Burgers equation

\[
\int_{\Omega} \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla)u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, dx = 0.
\]

- From the weak form of the PDE, we derive an equation to solve, that determines the state at each timestep in terms of the previous timestep

- Transcribe into Python – u is \(u^{n+1}\), u_ is \(u^n\):

```python
F = (inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v))))*dx
```
from firedrake import *
n = 50
mesh = UnitSquareMesh(n, n)

# We choose degree 2 continuous Lagrange polynomials. We also need a
# piecewise linear space for output purposes::
V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)

# We also need solution functions for the current and the next timestep::
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")
v = TestFunction(V)

# We supply an initial condition::
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)

# Start with current value of u set to the initial condition, and use the
# initial condition as our starting guess for the next value of u::
_u_.assign(ic)
u.assign(ic)

# :math:`\nu` is set to a (fairly arbitrary) small constant value::
nu = 0.0001
timestep = 1.0/n

# Define the residual of the equation::
F = inner((u - u_)/timestep, v) + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v))*dx

outfile = File("burgers.pvd")
outfile.write(project(u, V_out, name="Velocity"))

# Finally, we loop over the timesteps solving the equation each time::
t = 0.0
end = 0.5
while (t <= end):
solve(F == 0, u)
u_.assign(u)
t += timestep
outfile.write(project(u, V_out, name="Velocity"))

\[
\int_{\Omega} \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla) u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, dx = 0.
\]

- From the weak form of the PDE, we derive an equation to solve, that determines the state at each timestep in terms of the previous timestep
- Transcribe into Python – u is \( u^{n+1} \), \_u_ is \( u^n \):

\[
F = \left( \frac{(u - u_)}{timestep}, v \right) + \left( \nabla (u \cdot \nabla \text{grad}(u)) \right) \cdot v + \nu \left( \nabla u^{n+1} \cdot \nabla v \right) \, dx
\]

- **Burgers equation**

- **Firedrake implements the Unified Form Language (UFL)**

- **Embedded in Python**

- **UFL is also the DSL of the FEniCS project**

- What does its DSL actually look like?
```python
from firedrake import *
n = 50
mesh = UnitSquareMesh(n, n)

# We choose degree 2 continuous Lagrange polynomials.
# piecewise linear space for output purposes::
V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)

# We also need solution functions for the current and
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")
v = TestFunction(V)

# We supply an initial condition::
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)

# Start with current value of u set to
# initial condition as our starting guess
u_.assign(ic)
u.assign(ic)

# Define the residual of the equation::
F = (inner((u - u_)/timestep, v)
    + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx

nu = 0.000

timestep =

# Define t
F = (inner(u,nabla_grad(u)) + inner(u,nabla_grad(u)))*dx

# set up initial conditions for u and u_

end = 0.5
while (t <= end):
solve(F == 0, u)
u_.assign(u)
t += timestep

# Finally,
t = 0.0
end = 0.5
while (t <
solve(F == 0, u)
u_.assign(u)
t += timestep
outfile.write(project(u, V_out, name="Velocity"))
```

What does its DSL actually look like?
Generated code to assemble the resulting linear system matrix

Executed at each triangle in the mesh

Accesses degrees of freedom shared with neighbour triangles through indirection map
Firedrake: single-node AVX512 performance

Skylake cross-element vectorization

Does it generate good code?

A study of vectorization for matrix-free finite element methods, Tianjiao Sun et al
https://arxiv.org/abs/1903.08243
Firedrake: compiler architecture

- **UFL** specifies the (weak form of the) partial differential equation and how it is to be discretised
- **GEM**: abstract representation supports efficient flop-reduction optimisations
- **PyOP2**: stencil DSL for unstructured-mesh
- **Loo.py**: vectorization etc

**Sequence of intermediate representations**

- **100% Python, runtime code generation, code-caching**

---

**UFL**

- **PyOP2**
- **Loo.py**
- **Non-FE loops over the mesh**
- **Unified Form Language**
- **UFL “Two-stage” Form Compiler**
- **GEM: tensor contractions**
- **Loo.py representation**
- **PyOP2**
- **Loo.py loop transformations**
- **Distributed MPI-parallel PyOP2 implementation**
- **Multicore**
- **Manycore/GPU**
- **Future/other**

**Status**

- In production
- In development
- Some prototyping

---

Firedrake: a finite-element framework

- Automates the finite element method for solving PDEs
- Alternative implementation of FEniCS language, 100% Python using runtime code generation

**UFL** specifies the (weak form of the) partial differential equation and how it is to be discretised

- Compiler generates PyOP2 kernels and access descriptors

**GEM**: abstract representation supports efficient flop-reduction optimisations

- **PyOP2**: stencil DSL for *unstructured-mesh*
- Explicit *access descriptors* characterise access footprint of kernels

**Loo.py**: vectorization etc

Firedrake’s “Compiler architecture” has evolved over time
Example:

```c
for (i=0; i<N; ++i) {
    points[i]->x += 1;
}
```

Can the iterations of this loop be executed in parallel?

Oh no: not all the iterations are independent!

- You want to re-use piece of code in different contexts
- Whether it’s parallel depends on context!
Analysis is not always the interesting part....

It’s more fun the higher you start!

Compilation is like skiing

- Syntax
- Points-to
- Types
- Syntax

- Polyhedra
- Shape
- Dependence
- Call-graph
- Class-hierarchy

- Loop nest ordering
- Parallelisation

- Tiling
- Mapping
- Storage layout

- Instruction selection/scheduling
- Register allocation

http://www.nikkiemcdade.com/subFiles/2DExamples.html

http://www.ginz.com/new_zealand/ski_new_zealand_wanaka_cadrona
Unstructured meshes require pointers/indirection because adjacency lists have to be represented explicitly.

A controlled form of pointers (actually a general graph)

**OP2** is a C++ and Fortran library for parallel loops over the mesh, implemented by source-to-source transformation.

**PyOP2** is the same basic model, implemented in Python using runtime code generation.

Enables generation of highly-optimised vectorised, CUDA, OpenMP and MPI code.

The OP2 model originates from Oxford (Mike Giles et al).
How a mesh is represented in OP2

PyOP2:           "sets"               "dats"               "maps"

Mesh

Cells

Edges

Vertices

CellToEdge  EdgeToVertex
OP2 loops, access descriptors and kernels

**PyOP2:**
- **“sets”**
- **“dats”**
- **“maps”**

**op_par_loop(set, kernel, access descriptors)**

- **We specify which set to iterate over.**
- **We specify a kernel to execute** — the kernel operates entirely locally, on the **dats** to which it has access.
- **The access descriptors** specify which dats the kernel has access to:
  - Which dats of the target set
  - Which dats of sets indexed from this set through specified maps

- OP2 separates local (kernel) from global (mesh)
- OP2 makes data dependence explicit
PyOP2: “decoupled access-execute”

- Parallel loops, over sets (nodes, edges etc)
- Access descriptors specify how to pass data to and from the C kernel
- The kernel operates only on local data

```c
void res(float *A, float *u, float *du, const float *beta) {
    *du += (*beta) * (*A) * (*u);
}

void update(float *r, float *du, float *u, float *u_sum, float *u_max) {
    *u += *du + alpha * (*r);
    *du = 0.0f;
    *u_sum += (*u) * (*u);
    *u_max = *u_max > *u ? *u_max : *u;
}
```

for iter in xrange(0, NITER):
    u_sum = op2.Global(1, data=0.0, np.float32)
    u_max = op2.Global(1, data=0.0, np.float32)
    op2.par_loop(res, edges,
                  p_A(op2.READ),
                  p_u(op2.READ, edge2vertex[1]),
                  p_du(op2.INC, edge2vertex[0]),
                  beta(op2.READ))

    op2.par_loop(update, nodes,
                  p_r(op2.READ),
                  p_du(op2.RW),
                  p_u(op2.INC),
                  u_sum(op2.INC),
                  u_max(op2.MAX))
```

Access descriptors specify how to feed the kernel from the mesh.
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region
Code generation for indirect loops in PyOP2

For MPI we precompute partitions & haloes

Derived from PyOP2 access descriptors, implemented using PetSC DMPlax

At partition boundaries, the entities (vertices, edges, cells) form layered halo region

Core: entities owned which can be processed without accessing halo data.

Owned: entities owned which access halo data when processed

Exec halo: off-processor entities which are redundantly executed over because they touch owned entities

Non-exec halo: off-processor entities which are not processed, but read when computing the exec halo
Can we automate interesting optimisations that would be hard to do by hand?

First example:

- Tiling for cache locality

(This optimisation has been implemented – and automated – but does not currently form part of the standard distribution)
Sparse split tiling on an unstructured mesh, for locality

How can we load a block of mesh and do the iterations of loop 1, then the iterations of loop 2, before moving to the next block?

If we could, we could dramatically improve the memory access behaviour!
Partition the iteration space of loop 1

Colour the partitions, execute the colours in order

Project the tiles, using the knowledge that colour $n$ can use data produced by colour $n-1$

Thus, the tile coloured #1 grows where it meets colour #0

And shrinks where it meets colours #2 and #3
Partition the iteration space of loop 1

Colour the partitions

Project the tiles, using the knowledge that data produced by colour n can use data produced by colour n-1

Thus, the tile coloured #1 grows where it meets colour #0 and shrinks where it meets colours #2 and #3

Inspector-executor: derive tasks and task graph from the mesh, at runtime
Tiles grow

As we project the tiles forward, tile shape degrades
Perimeter-volume ratio gets worse
As we project the tiles forward, tile shape degrades. Perimeter-volume ratio gets worse. We could partition Loop 1’s data for the cache. But Loop 2 and Loop 3 have different footprints. So we rely on good (ideally space-filling-curve) numbering.
Tiles can collide

(1) Blue, (2) Red, (3) Green

Loop chains

```python
with loop_chain(tile_size=..., ...):
    # solve for velocity vector field
    self.solve(...);
    self.solve(...);
    self.solve(...);
    self.solve(...);

    # In case the source is time-dependent, update the time 't' here.
    if(self.source):
        with timed_region('source term update'):
            self.source_expression.t = t
            self.source = self.source_expression

    # Solve for the velocity vector field.
    self.solve(self.rhs_uh1, self.velocity_mass_asdat, self.uh1)
    self.solve(self.rhs_stemp, self.stress_mass_asdat, self.stemp)
    self.solve(self.rhs_uh2, self.velocity_mass_asdat, self.uh2)
    self.solve(self.rhs_u1, self.velocity_mass_asdat, self.u1)

    # Solve for the stress tensor field.
    self.solve(self.rhs_sh1, self.stress_mass_asdat, self.sh1)
    self.solve(self.rhs_utemp, self.velocity_mass_asdat, self.utemp)
    self.solve(self.rhs_sh2, self.stress_mass_asdat, self.sh2)
    self.solve(self.rhs_s1, self.stress_mass_asdat, self.s1)

    self.u0.assign(self.u1)
    self.s0.assign(self.s1)

# Write out the new fields
self.write(self.u1, self.s1, self.tofile and timestep % self.output == 0)

# Move onto next timestep
s = self.dt
timestep += 1
```

(Luporini, Lange, Jacobs, Gorman, Ramanujam, Kelly. Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling. ACM TOMS 2019
https://doi.org/10.1145/3302256)
Example: Seigen

- Elastic wave solver
- 2d triangular mesh
- Velocity-stress formulation
- 4th-order explicit leapfrog timestepping scheme
- Discontinuous-Galerkin, order q=1-4
- 32 nodes, 2x14-core E5-2680v4, SGI MPT 2.14
- 1000 timesteps (ca.1.15s/timestep)

Up to 1.28x speedup

Inspection about as much time as 2 timesteps

Using RCM numbering – space-filling curve should lead to better results

Best speedup: 1.28x at q=3 on 448 processes.

Optimum fusion scheme breaks 25 loops into 6 chains. MPI halo is extended from S=1 to S=2
Can we automate interesting optimisations that would be hard to do by hand?

Second example:

Generalised loop-invariant code motion

(This optimisation has been implemented, automated, and re-implemented – and forms part of the standard distribution)
Generated code to assemble the resulting linear system matrix

Executed at each triangle in the mesh

Accesses degrees of freedom shared with neighbour triangles through indirection map
Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.

The local assembly operation computes a small dense submatrix

These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

void helmholtz(double A[3][3], double **coords) {
    // K, det = Compute Jacobian (coords)

    static const double W[3] = {...}
    static const double X_D10[3][3] = {...}
    static const double X_D01[3][3] = {...}

    for (int i = 0; i<3; i++)
        for (int j = 0; j<3; j++)
            for (int k = 0; k<3; k++)
                A[j][k] += ((Y[i][k]*Y[i][j] +
                                ((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j])) +
                                ((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
                                *det*W[i]);
}
Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.

The local assembly operation computes a small dense submatrix.

These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE.

```c
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
                     +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
                     +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*det*W[i];
}
```
Generalised loop-invariant code motion:

```c
void helmholtz(double A[3][4], double **coords) {
#define ALIGN __attribute__((aligned(32)))
// K, det = Compute Jacobian (coords)

static const double W[3] ALIGN = {...}
static const double X_D10[3][4] ALIGN = {{...}}
static const double X_D01[3][4] ALIGN = {{...}}

for (int i = 0; i<3; i++) {
    double LI_0[4] ALIGN;
    double LI_1[4] ALIGN;
    for (int r = 0; r<4; r++) {
        LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
        LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
    }
    for (int j = 0; j<3; j++)
        #pragma vector aligned
        for (int k = 0; k<4; k++)
            A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i]);
}
```
We formulate an ILP problem to find the best factorisation strategy.
FOCUS ON HYPERELASTICITY

Firedrake’s “Compiler architecture” has evolved over time

Non-FE loops over the mesh
Unified Form Language
UFL “Two-stage” Form Compiler
PyOP2

Distributed MPI-parallel PyOP2 implementation
COFFEE kernel optimiser/vectoriser

PyOP2

Distributed MPI-parallel PyOP2 implementation

Loo.py representation

GEM: tensor contractions

Loo.py loop transformations

Vectorisation

Multicore
Manycore /GPU
Future/other

In production
In development
Some prototyping

Loop-invariant code motion, sum-factorisation
Engaging with applications to exploit domain-specific optimisations can be incredibly fruitful

Compiling general purpose languages is worthy but usually incremental

Compiler architecture is all about designing intermediate representations – that make hard things look easy

Tools to deliver domain-specific optimisations often have domain-specific representations

Premature lowering is the constant enemy (appropriate lowering is great)

Along the way, we learn something about building better general-purpose compilers and programming abstractions

Drill vertically, expand horizontally
Sparse unstructured tiling really works, but didn’t make it into the main trunk

- It’s just too complicated to justify the additional maintenance burden
- It only helps some applications
- We need to find a way to make it easier!

- Improved strong-scaling
- GPUs (and other accelerators?)
- Coupled problems (in-progress)
- Particles, particle transport
- Mesh adaptation, load balancing

**Things that I haven’t had time to talk about:**

- Automatic adjoints, inverse problems (in-service)
- Interface/integration with PetSc (in-service)
- Hybridisation, static condensation (in-service, could be faster)
The real value of Firedrake is in supporting the applications users in exploring their design space.

We enable them to navigate rapidly through alternative solutions to their problem.

We break down barriers that prevent the right tool being used for the right problem.

Firedrake automates the finite element method.

The Devito project automates finite difference.

In the future, we will have automated pathways from maths to code for many classes of problem, and many alternative solution techniques.
We can simultaneously raise the level at which programmers can reason about code, provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand.

Program generation is how we do it.
Partly funded/supported by

- NERC Doctoral Training Grant (NE/G523512/1)
- EPSRC “MAPDES” project (EP/I00677X/1)
- EPSRC “PSL” project (EP/I006761/1)
- Rolls Royce and the TSB through the SILOET programme
- EPSRC “PAMELA” Programme Grant (EP/K008730/1)
- EPSRC “PRISM” Platform Grants (EP/I006761/1 and EP/R029423/1)
- EPSRC “Custom Computing” Platform Grant (EP/I012036/1)
- EPSRC “Application Customisation” Platform Grant (EP/P010040/1)
- EPSRC “A new simulation and optimisation platform for marine technology” (EP/M011054/1)
- Basque Centre for Applied Mathematics (BCAM)

Code:

- [http://www.firedrakeproject.org/](http://www.firedrakeproject.org/)
- [http://op2.github.io/PyOP2/](http://op2.github.io/PyOP2/)
- [https://github.com/OP-DSL/OP2-Common](https://github.com/OP-DSL/OP2-Common)