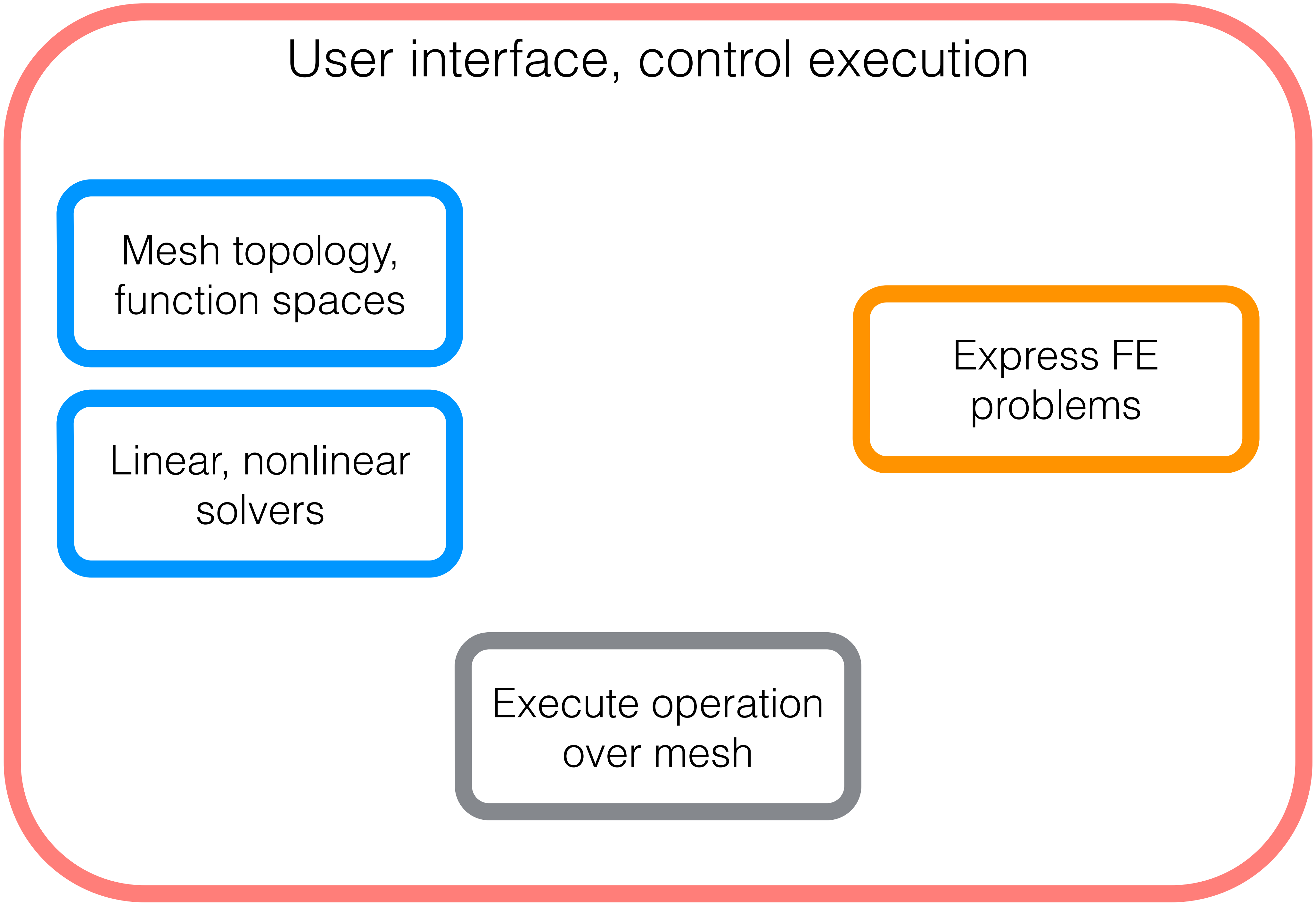
Firedrake/PyOP2: an overview

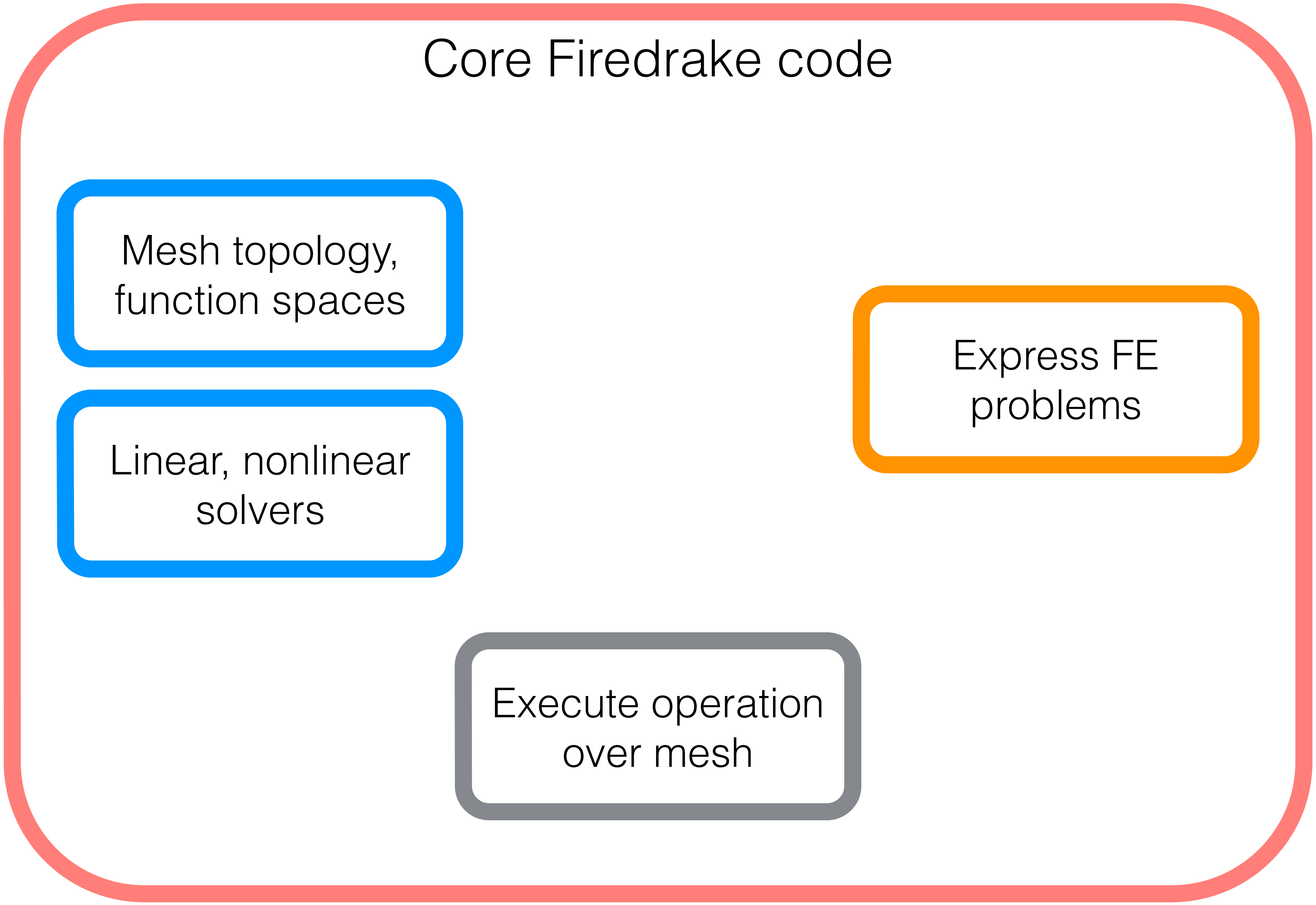
Lawrence Mitchell (lawrence.mitchell@imperial.ac.uk)

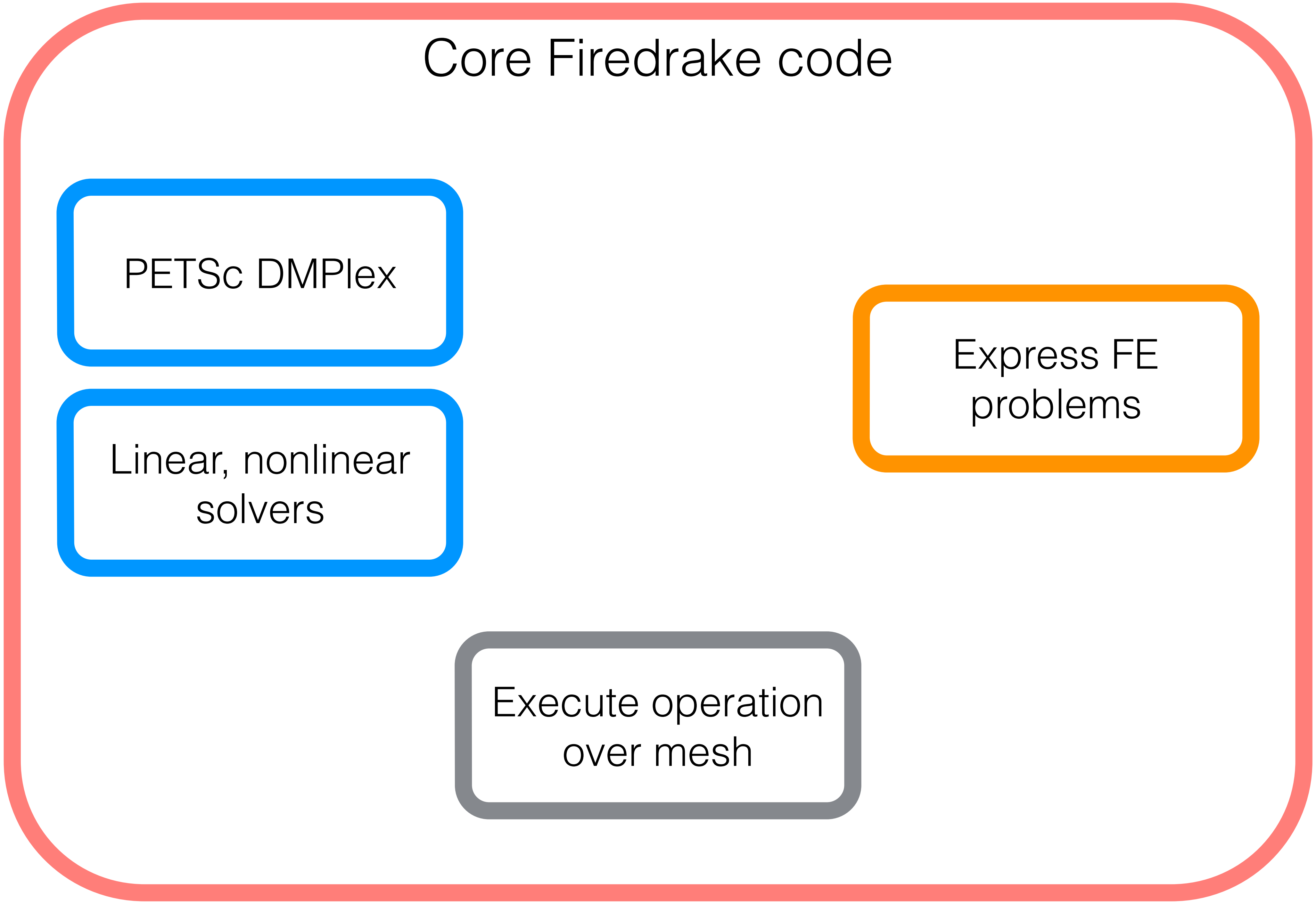
- www.firedrakeproject.org
- github.com/OP2/PyOP2
- github.com/firedrakeproject/firedrake
- firedrake@imperial.ac.uk (subscribe first)
- #firedrake on irc.freenode.net
- WPL upstairs (modulo summer building works)

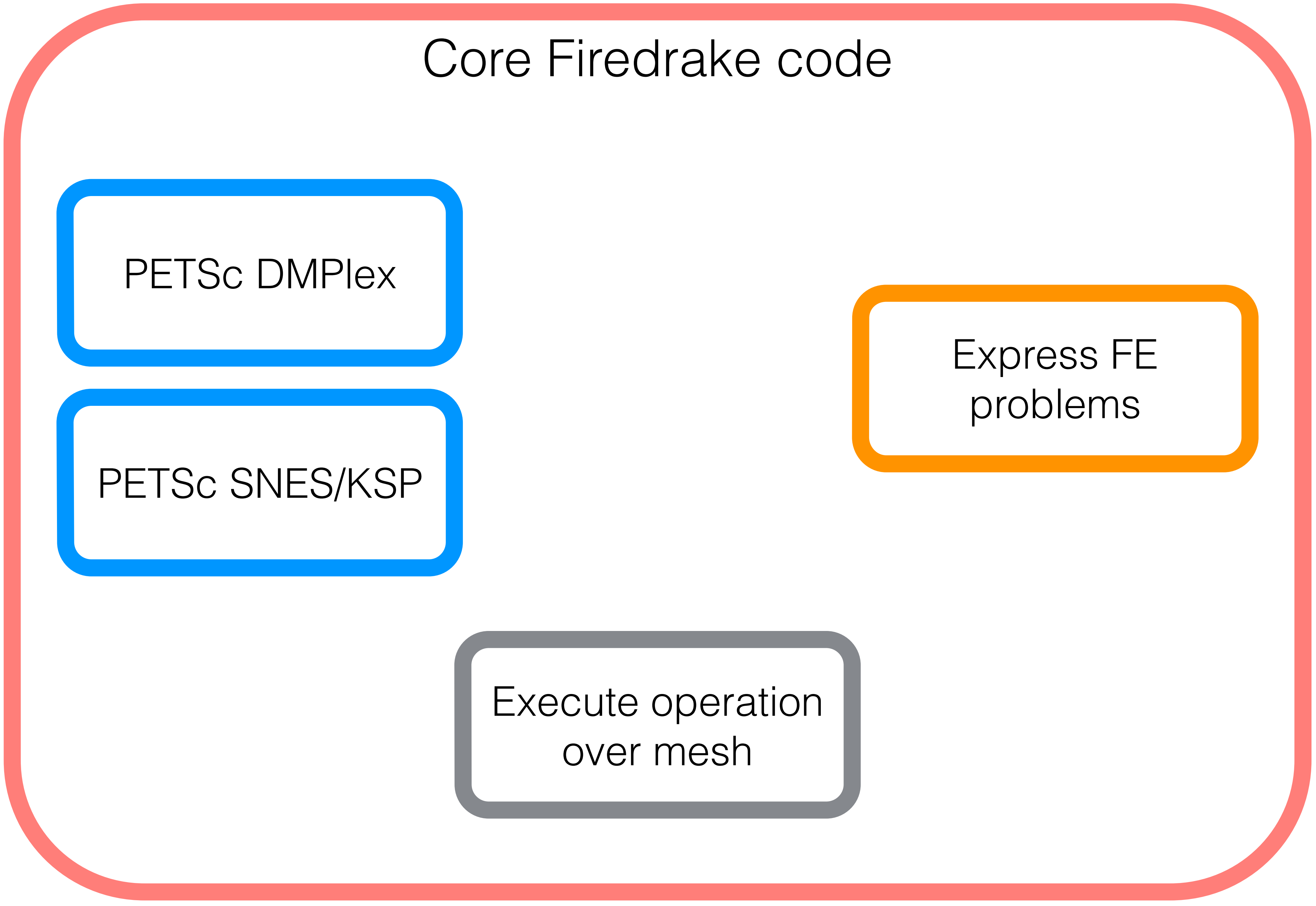
...and a cast of thousands

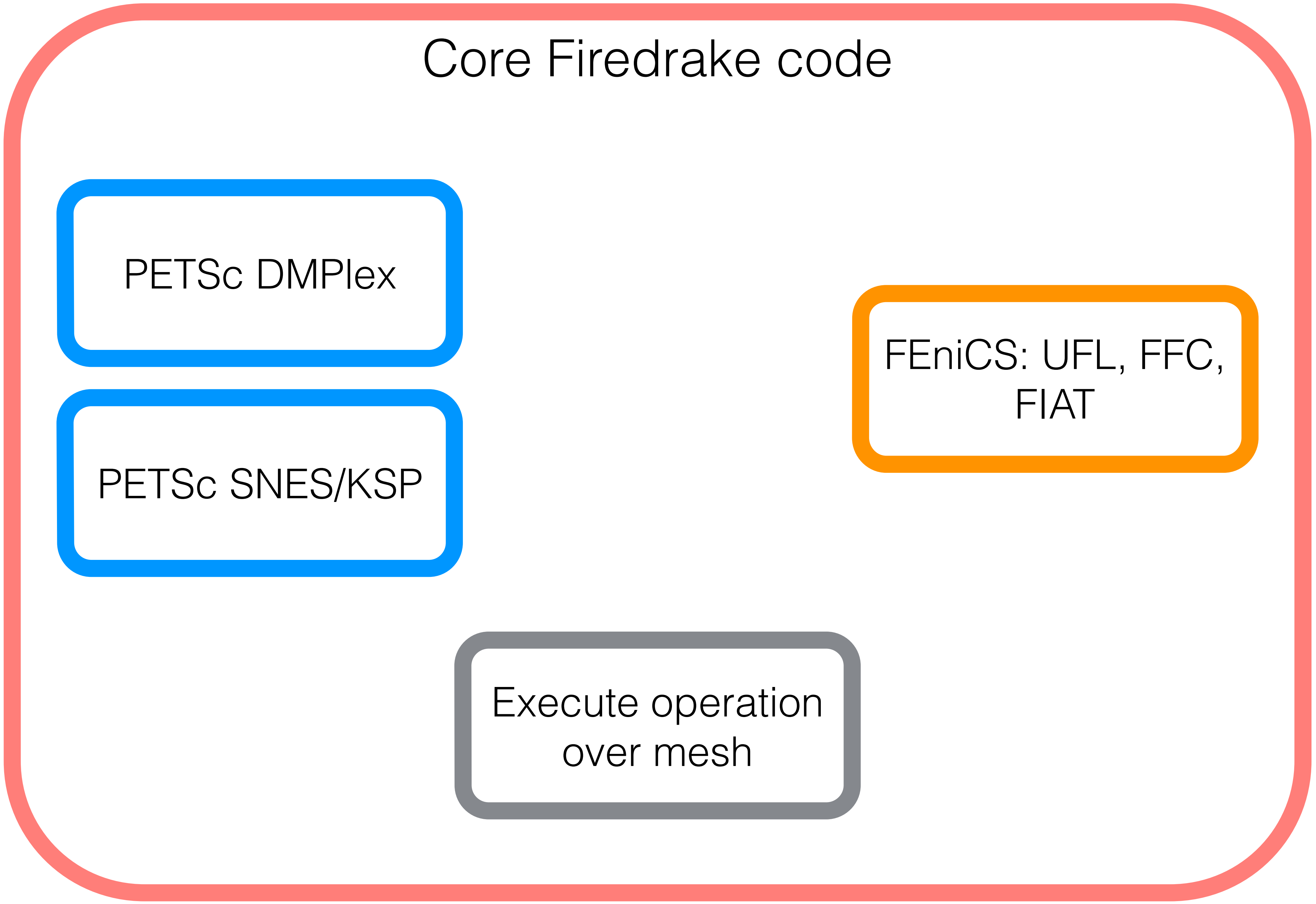
- Computing
  - Doru Bercea, Fabio Luporini, Florian Rathgeber, Lawrence Mitchell, David Ham, Paul Kelly
- Maths
  - Andrew McRae, Colin Cotter, David Ham, Jemma Shipton, Hiroe Yamazaki
- ESE
  - Michael Lange, Christian Jacobs
- Bath
  - Eike Müller
- Simula
  - Simon Funke
- Former members
  - Graham Markall, Nicolas Loriant
- UROP/project students
  - Kaho Sato, George Boutsioukis


User interface, control execution
Mesh topology, function spaces
Linear, nonlinear solvers
Express FE problems
Execute operation over mesh

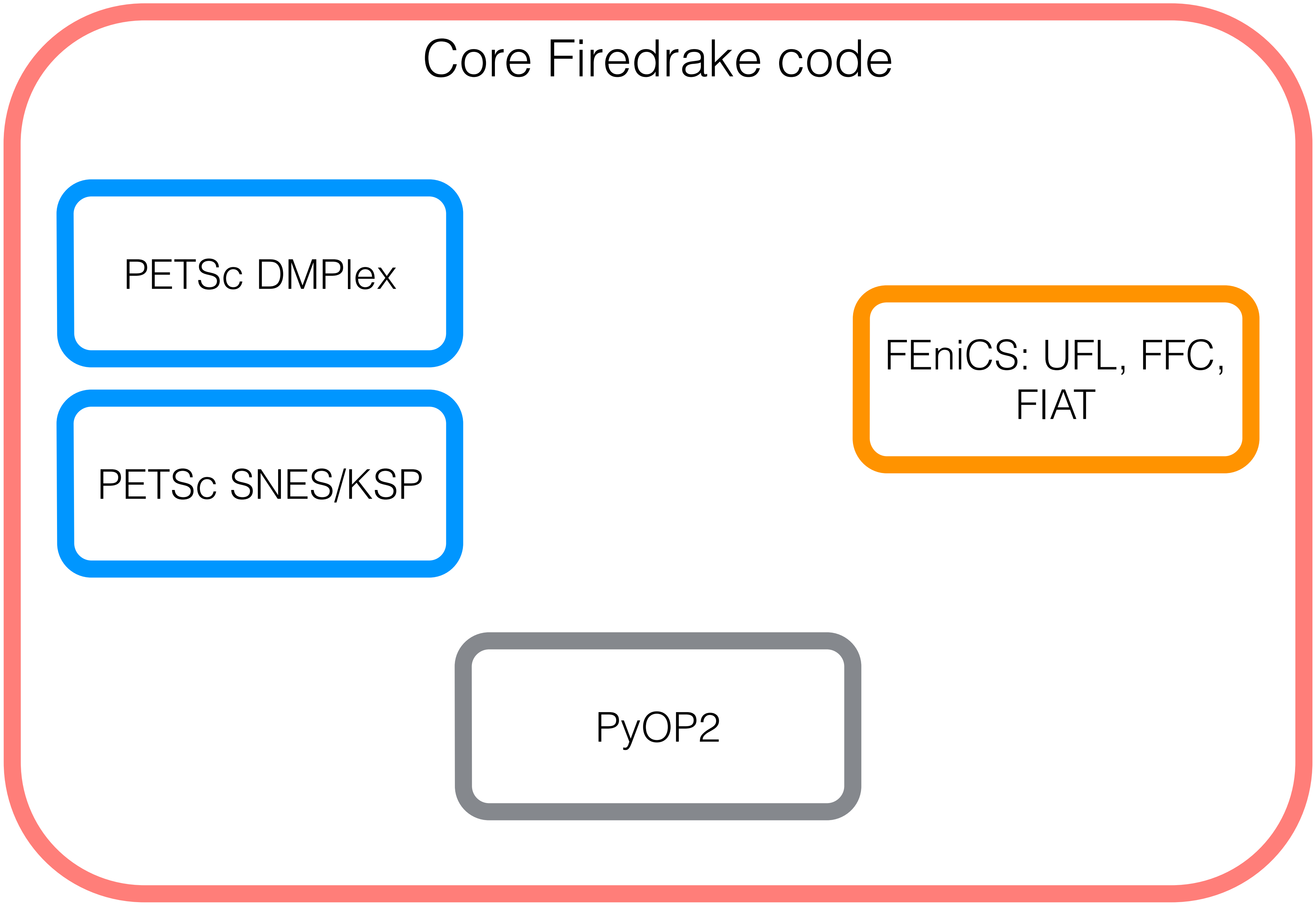

Core Firedrake code
Mesh topology,
function spaces
Linear, nonlinear
solvers
Express FE
problems
Execute operation
over mesh


Core Firedrake code
PETSc DMPlex
Linear, nonlinear solvers
Express FE problems
Execute operation over mesh


Core Firedrake code
PETSc DMPlex
PETSc SNES/KSP
Express FE problems
Execute operation over mesh



Core Firedrake code
PETSc DMPlex
PETSc SNES/KSP
FEniCS: UFL, FFC, FIAT
Execute operation over mesh


Core Firedrake code
PETSc DMPlex
PETSc SNES/KSP
FEniCS: UFL, FFC, FIAT
PyOP2

```python
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

```python
from firedrake import *
m = Mesh('...')
mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

Michael's talk (DMPlex)

```python
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

Extrusion

```python
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

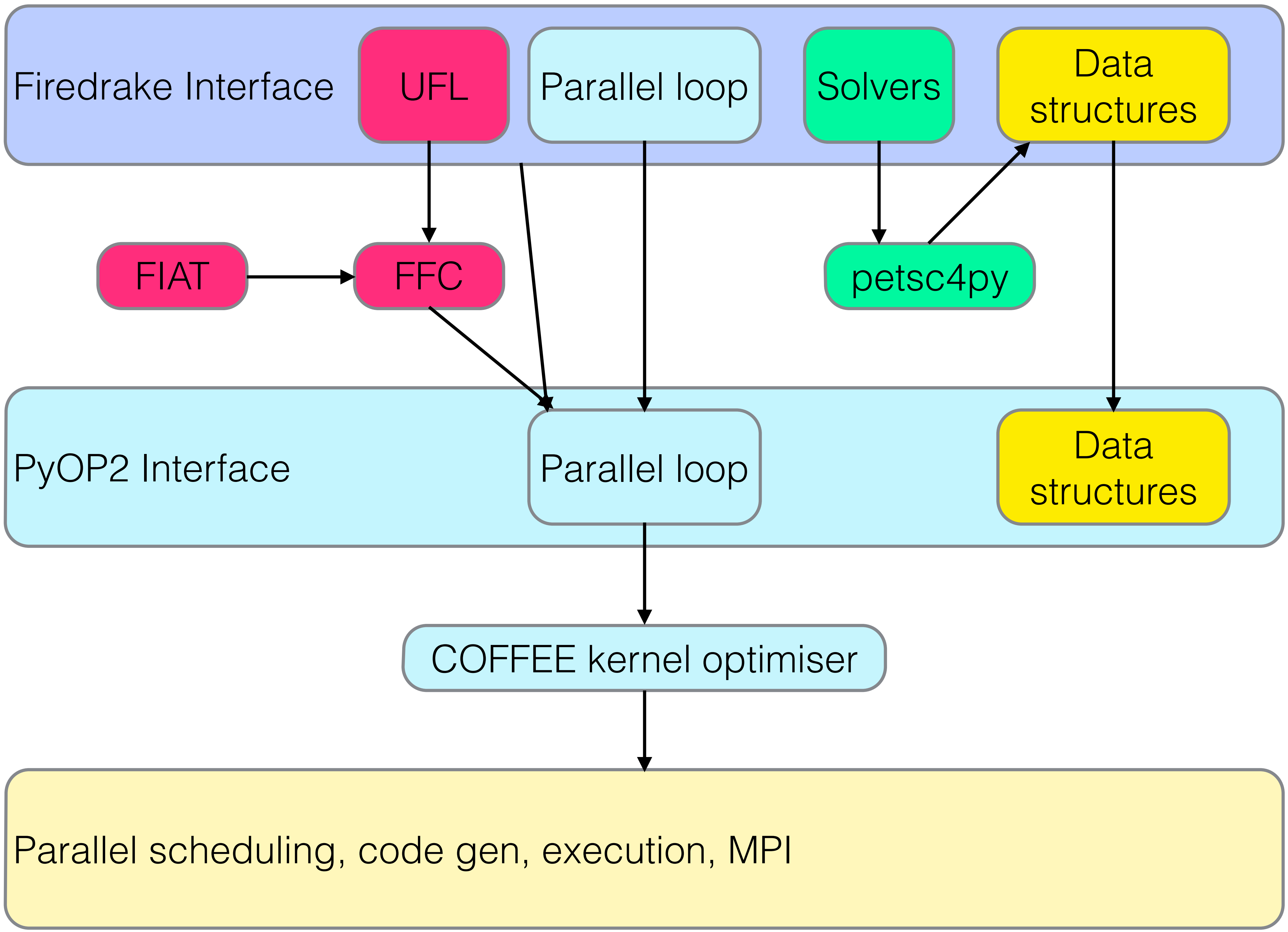
UFL

```python
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

← Firedrake

```python
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)
solve(a == L, ...)
```

Firedrake, PyOP2, FFC, FIAT, UFL, PETSc


Firedrake Interface
UFL
Parallel loop
Solvers
Data structures
FIAT
FFC
petsc4py
PyOP2 Interface
Parallel loop
Data structures
COFFEE kernel optimiser
Parallel scheduling, code gen, execution, MPI

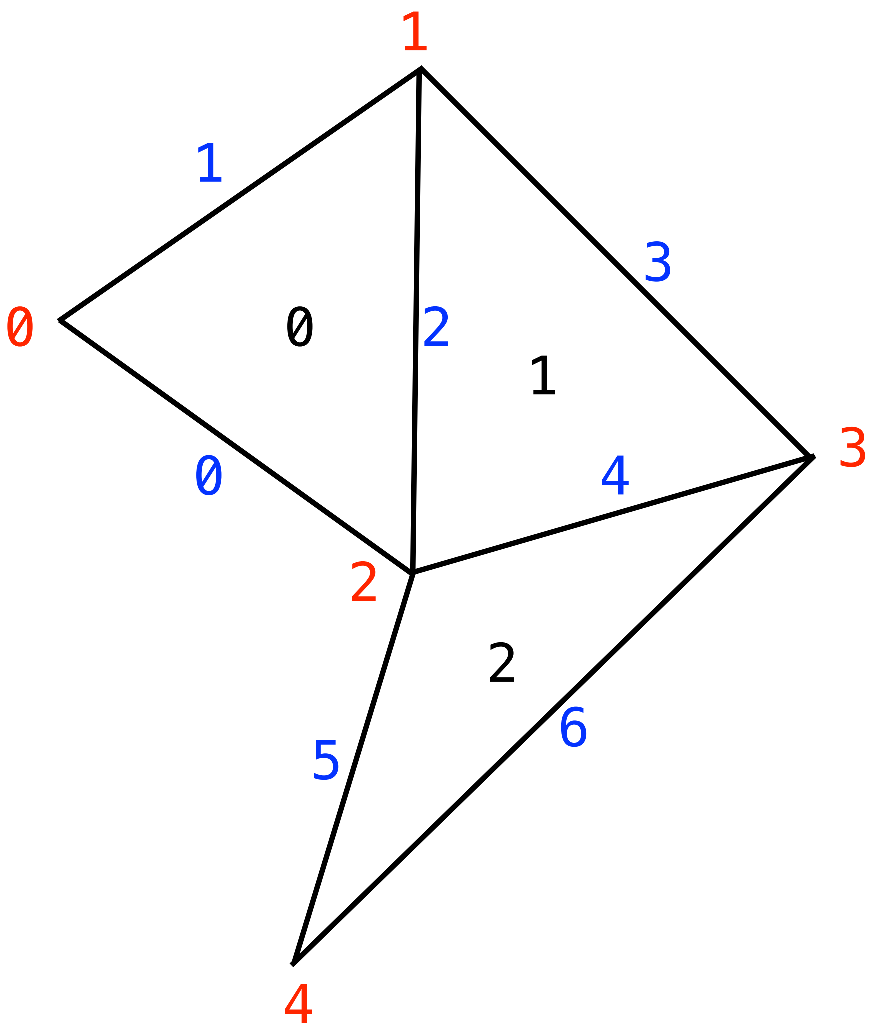
PyOP2

- Computation on a fixed-degree graph (unstructured mesh)
- Local computation *kernel* executed everywhere
    - Data parallel
- *Reductions* aggregate contributions into result
- No finite element assumption (almost)

Mesh topology

- Effectively a graph
- Sets represent nodes in the graph
- Maps define edges between nodes
- Maps are constant *arity*
    - Cannot have map from vertices to cells (say)

```python
from pyop2 import op2
op2.init()

cells = op2.Set(3)
edges = op2.Set(7)
vertices = op2.Set(5)

c2e = op2.Map(cells, edges, 3,
              [[0, 1, 2],
               [2, 3, 4],
               [4, 5, 6]])
c2v = op2.Map(cells, vertices, 3,
              [[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
e2v = op2.Map(edges, vertices, 2,
              [[0, 2],
               [0, 1],
               [1, 2],
               [1, 3],
               [2, 3],
               [2, 4],
               [3, 4]])
```



Data

- Data defined on DataSets (Set + number of entries per set entity), any C type

```
from pyop2 import op2
op2.init()
cells = op2.Set(3)
dofs = op2.Set(18)
```

```
field = op2.Dat(dofs**1, dtype=np.float64)
```

1 dof per set entity

```
vector_field = op2.Dat(dofs**2, dtype=np.float32)
```

2 dofs per set entity

Kernel

- Acts on *local* data, cannot read outside it

```
from pyop2 import op2
op2.init()

edges = op2.Set(...)
vertices = op2.Set(...)

midpoint = op2.Dat(edges**2)
coords = op2.Dat(vertices**2)

e2v = op2.Map(edges, vertices, 2, ...)

midpoint_kernel = op2.Kernel("""
     void midpoint(double *mid, double **vtx_coords) {
         mid[0] = (vtx_coords[0][0] + vtx_coords[1][0]) / 2;
         mid[1] = (vtx_coords[0][1] + vtx_coords[1][1]) / 2;
     }""", "midpoint")
```

Parallel loop

- Execute *kernel* over set, reading/writing data
  - Describe: iteration space (set/subset), data to be accessed, access type (READ, WRITE, RW, INC), indirection.
- MPI-collective
  - All processes must hit parallel loop (even if they are iterating over zero-sized domain)
    - Careful applying boundary conditions!
  - Kernel must be *identical* everywhere
    - Run with `export PYOP2_DEBUG=1` to find issues

```python
from pyop2 import op2
op2.init()

edges = op2.Set(...)
vertices = op2.Set(...)

midpoint = op2.Dat(edges**2)
coords = op2.Dat(vertices**2)

e2v = op2.Map(edges, vertices, 2, ...)

midpoint_kernel = ...

op2.par_loop(midpoint_kernel, edges,
             midpoint(op2.WRITE),
             coords(op2.READ, e2v))
```

Direct write to midpoint

Indirect read from coords through e2v

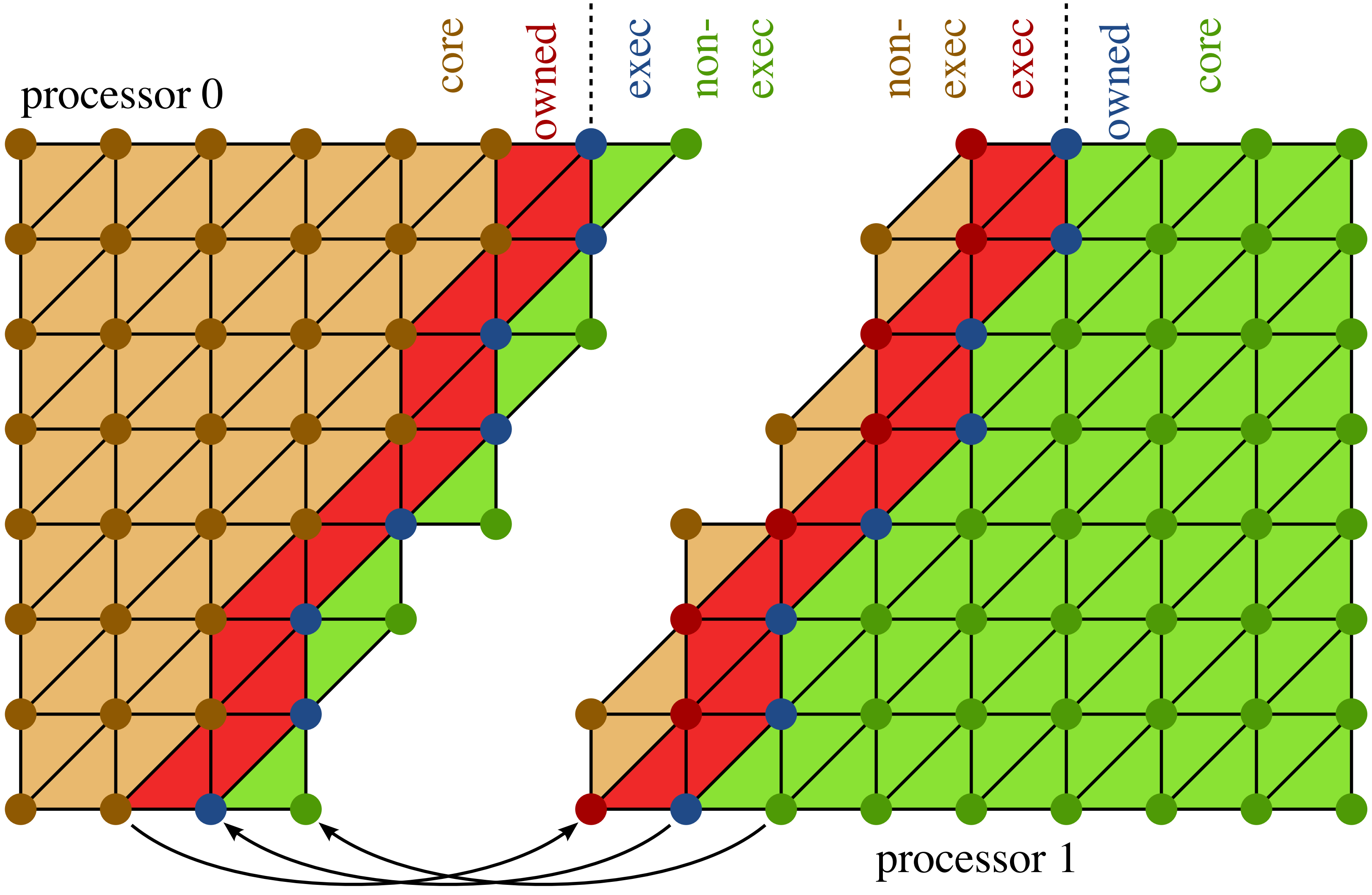
Code gen

- API carefully designed to make automated reasoning about execution *easy* "synthesis, not analysis"
- Access descriptors allow automatic reasoning about data dependencies
- e.g. no need to halo exchange on WRITE data
    - But mark as dirty so subsequent reads force exchange
- MPI-collective semantics mean we don't need to worry about other processes not posting halo sends/receives
- Shared memory: colour for non-conflicting execution
    - Assumption: we're allowed to reorder computations

```python
op2.par_loop(midpoint_kernel, edges,
             midpoint(op2.WRITE),
             coords(op2.READ, e2v))
```

```c
void midpoint(double *mid, double **vtx_coords) {
   mid[0] = (vtx_coords[0][0] + vtx_coords[1][0]) / 2;
   mid[1] = (vtx_coords[0][1] + vtx_coords[1][1]) / 2;
}

void wrap_midpoint(int start, int end,
                   double *midpoint, double *coords, int *e2v) {
   double *vtx_coords[2];
   for ( int i = start; i < end; i++ ) {
       vtx_coords[0] = coords + (e2v[i * 2 + 0])*2;
       vtx_coords[1] = coords + (e2v[i * 2 + 1])*2;
       midpoint(midpoint + i * 2, vtx_coords);
   }
}
```

- MPI parallelism handled in Python
- Entities must be *ordered*, extents can encode iteration order. Effectively, this puts constraint on numbering in Maps.

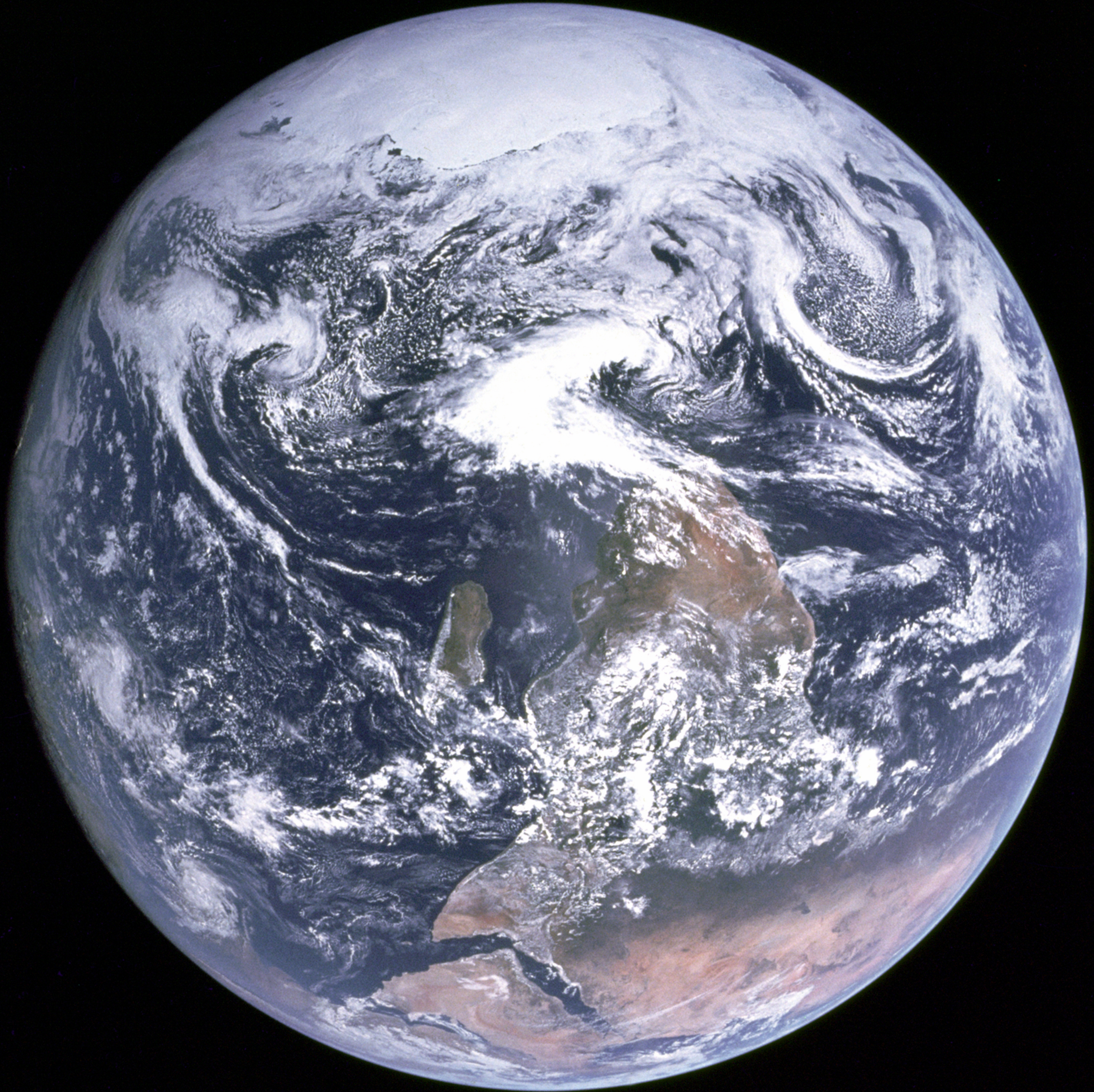


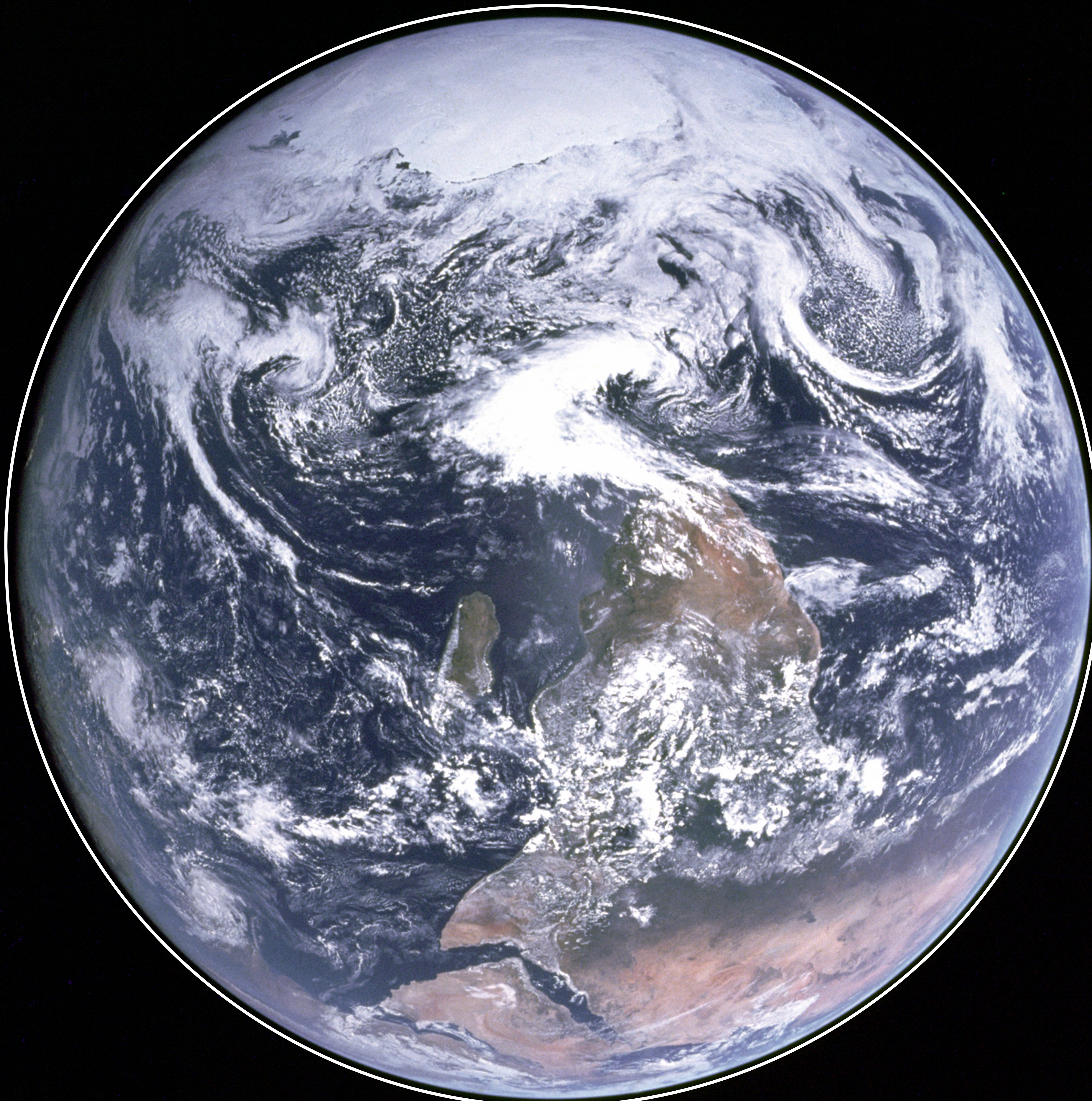
- *Core* entries: can compute without up to date halos
- *Owned* entries: local data, needs up to date halos
- *Exec halo* entries: remote data, but writes (through a map) to local data
- *Non-exec halo* entries: remote data, read (through a map) when computing on exec halo
- For FE, the stencil *on the topology* tells us how to mark topological entities, and hence degrees of freedom

```
cells = op2.Set([1, 3, 5, 6])
dofs = op2.Set([43, 52, 63, 68])

c2d = op2.Map(cells, dofs, 3,
             [
              # core entry (points to within first 43 dofs)
              [0, 32, 40],
              # owned entries (point to within first 52 dofs, but not
              # all within first 43)
              [0, 43, 44],
              [1, 45, 51],
              # exec halo entries (point to within first 63 dofs, but
              # not all within first 52)
              [2, 52, 57],
              [55, 56, 62],
              # non-exec halo entries (anything else)
              [63, 64, 65]
             ])
```

Extruded meshes

PyOP2 support



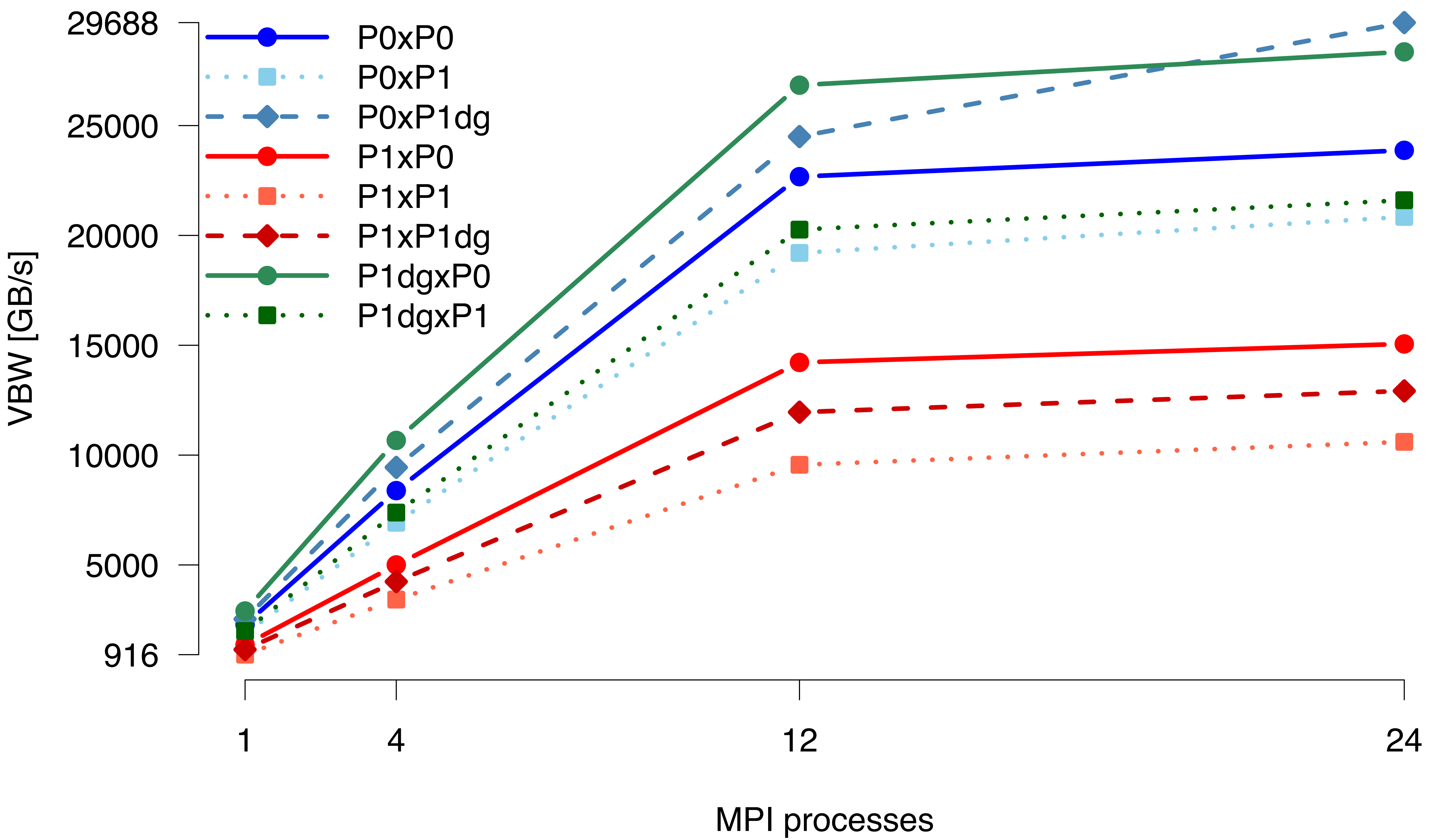


- Structured in *vertical* direction
- Map implicit (just provide *base* map plus offsets)
- Amortizes cost of indirect lookups
  - 70% peak memory bandwidth
- We measure *valuable bandwidth*: assume we could move data *perfectly* and therefore only ever touch it once. A *lower bound* on actual data movement.

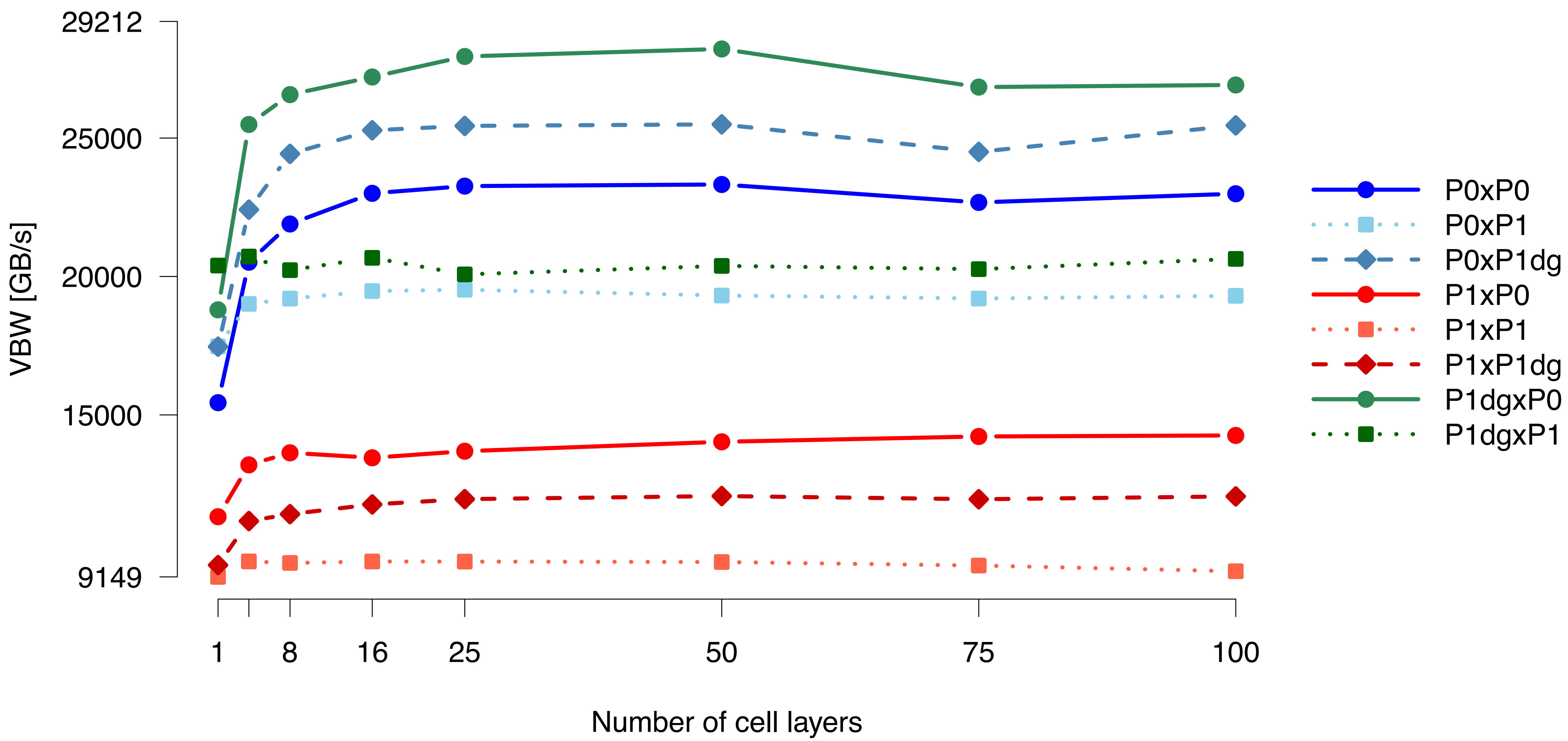
Does it work?

12 core SandyBridge; STREAM triad 42 GB/s

`assemble(v*dx)`



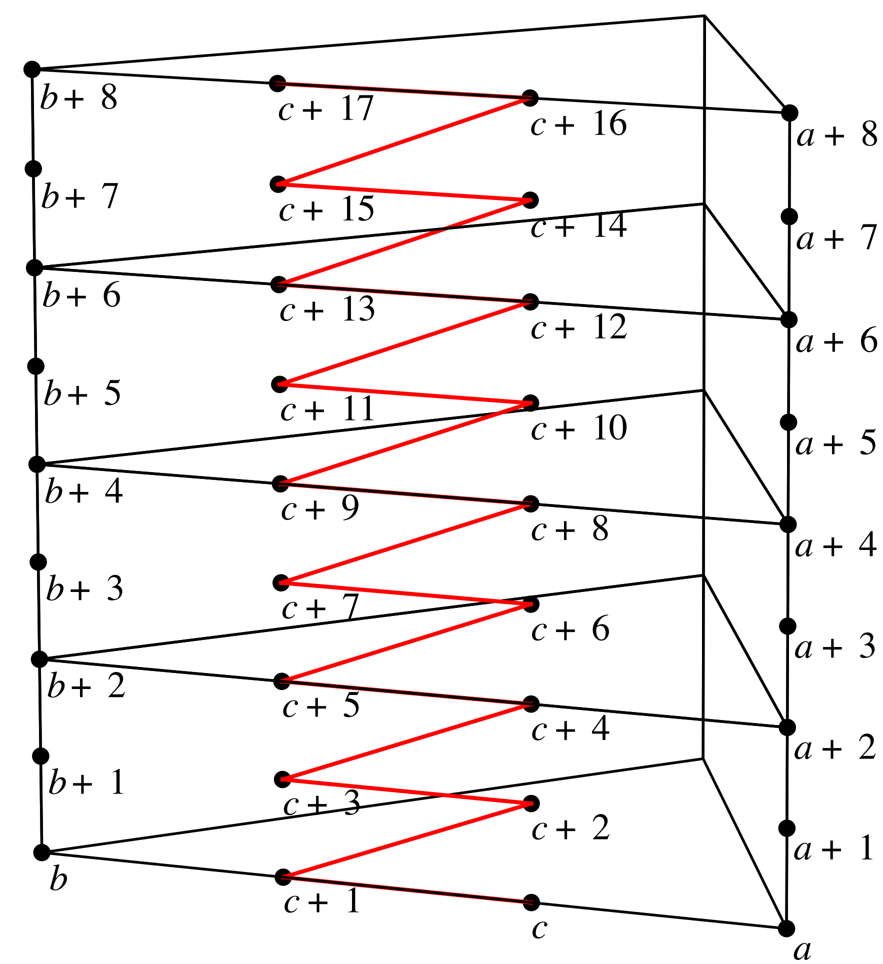
Indirection amortized after around 20 layers



How does it work?

- Indirect lookup for base cell dofs
- Direct increment walking up column
- Extra PyOP2 types
- Sets carry number of layers
- Maps carry offset for each indirected dof

```
base_cells = op2.Set(1)
ext_cells = op2.ExtrudedSet(base, layers=5)
base_dofs = op2.Set(total_dofs)
ext_dofs = op2.ExtrudedSet(base_dofs, layers=5)

cell2dof = op2.Map(ext_cells, ext_dofs,
                  12,
                  vals=[a, a+1, a+2, b, b+1, b+2,
                   c, c+1, c+2, c+3, c+4, c+5],
                  offset=[2, 2, 2, 2, 2, 2,
                          4, 4, 4, 4, 4, 4])
```



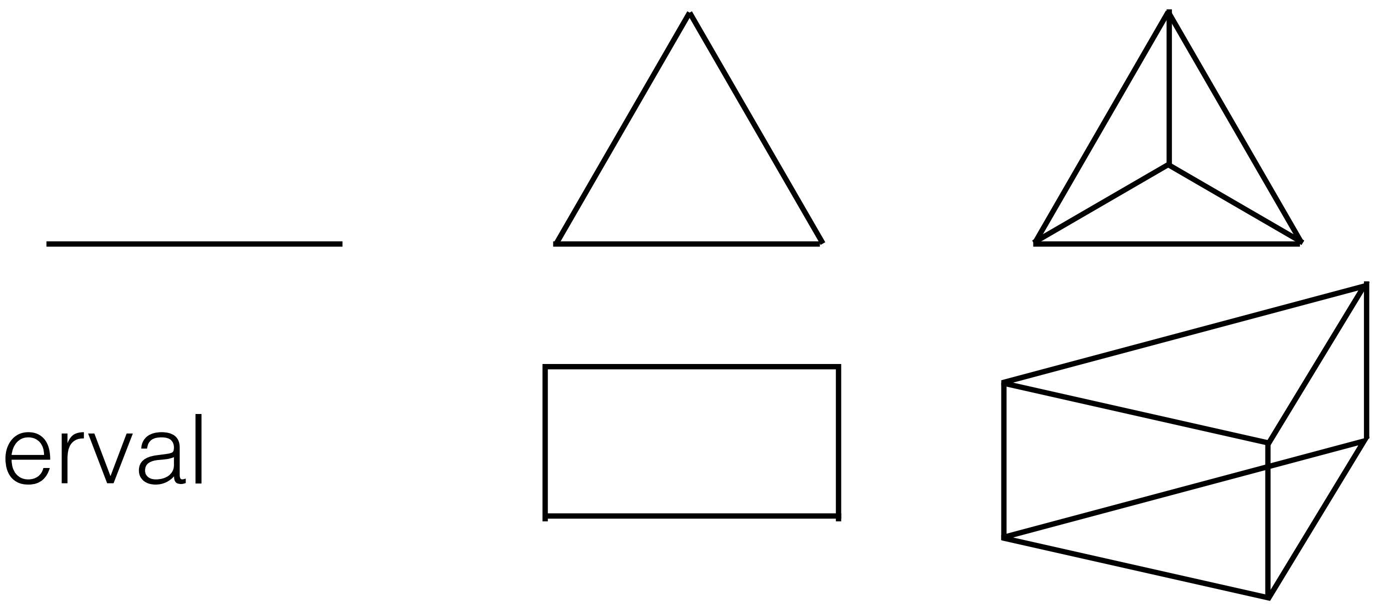
```c
void uniform_extrusion_kernel(double **base_coords, double **ext_coords,
                              int **layer, double *layer_height) {
   for ( int d = 0; d < 3; d++ ) {
       for ( int c = 0; c < 2; c++ ) {
           ext_coords[2*d][c] = base_coords[d][c];
           ext_coords[2*d+1][c] = base_coords[d][c];
       }
       ext_coords[2*d][2] = *layer_height * (layer[0][0]);
       ext_coords[2*d+1][2] = *layer_height * (layer[0][0] + 1);
   }
}
void wrap_uniform_extrusion_kernel(int start, int end, double *base_coords,
                                  int *base_coords_map, double *ext_coords,
                                  int *ext_coords_map, int *layer,
                                  int *layer_map, double *layer_height,
                                  int start_layer, int end_layer) {
   double *base_packed[3];
   double *ext_packed[6];
   int *layer_packed[1];
   for ( int i = start; i < end; i++ ) {
       base_packed[0] = base_coords + (base_coords_map[i * 3 + 0])* 2;
       base_packed[1] = base_coords + (base_coords_map[i * 3 + 1])* 2;
       base_packed[2] = base_coords + (base_coords_map[i * 3 + 2])* 2;
       ext_packed[0] = ext_coords + (ext_coords_map[i * 6 + 0])* 3;
       ext_packed[1] = ext_coords + (ext_coords_map[i * 6 + 1])* 3;
       ext_packed[2] = ext_coords + (ext_coords_map[i * 6 + 2])* 3;
       ext_packed[3] = ext_coords + (ext_coords_map[i * 6 + 3])* 3;
       ext_packed[4] = ext_coords + (ext_coords_map[i * 6 + 4])* 3;
       ext_packed[5] = ext_coords + (ext_coords_map[i * 6 + 5])* 3;
       layer_packed[0] = layer + (layer_map[i * 1 + 0])* 1;
       for (int j = start_layer; j < end_layer; ++j) {
           uniform_extrusion_kernel(base_packed, ext_packed, layer_packed, layer_height);
           ext_packed[0] += 3;
           ext_packed[1] += 3;
           ext_packed[2] += 3;
           ext_packed[3] += 3;
           ext_packed[4] += 3;
           ext_packed[5] += 3;
           layer_packed[0] += 1;
       }
   }
}
```

Platform portability

- Theory: PyOP2 code is portable across hardware platforms (CPU/GPU) since intent of iteration is separated from its implementation
- Some PyOP2 codes work on both CPU and GPU
- FEM kernels need different implementations on CPU/GPU (practice)
  - CPU: assemble multiple element tensors per thread
  - GPU: one element tensor *entry* per thread
- Reality: PyOP2 offers *potential* for platform portability, but is currently single (CPU (+ shared memory)) platform for practical purposes.

Firedrake

- Simplices
- Simplex x interval



- H^1, H(div), H(curl) and L^2 discretisations
- Mixed finite elements
- Parallel (MPI [+ OpenMP]) via PyOP2

No computation

- Firedrake carries out (almost) zero computation itself
  - Constructing dof maps for function spaces is our job (not really an abstraction available here)
- All solving is devolved to PETSc
  - Abstraction: non-linear solves require residual evaluation and matrix-vector products (of the linearisation of the residual)
- All FE assembly devolved to PyOP2
  - Abstraction: FE assembly is the evaluation of a *local* kernel repeatedly over a mesh of *global* data

(Almost) no code

```
$ cloc firedrake/
      30 text files.
      30 unique files.
       1 file ignored.

http://cloc.sourceforge.net v 1.60  T=0.17 s (168.6 files/s, 50255.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          27           1359           1865           4282
Cython                           1            107            138            620
C/C++ Header                     1             33             37            204
-------------------------------------------------------------------------------
SUM:                            29           1499           2040           5106
-------------------------------------------------------------------------------
```

Why?

- Use existing solutions wherever possible
- Militant about sticking to abstractions
    - The only way to do mesh-size operations is with PyOP2 loops
    - Mesh setup an exception, it is by far the most difficult bit of code

Behind the solve call

```
from firedrake import *

m = Mesh('...')

mesh = ExtrudedMesh(m, layers=20, extrusion_type='uniform')

element1 = ...
element2 = ...

V = FunctionSpace(mesh, element1)
Q = FunctionSpace(mesh, element2)

W = V*Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)

a = fn(u, p, v, q)
L = fn(v, q)

solve(a == L, ...)
```

```
solve(a == L, u, ...)
```

- Always solve using nonlinear solver
  - Jacobian is `a`
  - Residual is `action(a, u) - L =: F`
  - Solver is a PETSc SNES
    - provide residual evaluation `assemble(F)`
    - and Jacobian `assemble(a)`

Inside assemble

- Call ffc to turn form into PyOP2 kernels (COFFEE AST)
- inspect form to determine coefficients and any (maybe) trial and test functions
- Construct arguments to PyOP2 par_loop
- Call par_loop

```python
for integral_type, coords, coefficients, kernel in kernels:
   m = coords.function_space().mesh()
   if integral_type == 'cell':
       if is_mat:
           tensor_arg = mat_tensor(op2.INC,
                                   (s.cell_node_map()[op2.i[0]],
                                    s.cell_node_map()[op2.i[1]]))
       elif is_vec:
           tensor_arg = vec_tensor(op2.INC,
                                   s.cell_node_map()[op2.i[0]])
       else:
           tensor_arg = tensor(op2.INC)

       itspace = m.cell_set
       itspace._extruded_bcs = extruded_bcs
       args = [kernel, itspace, tensor_arg,
               coords.dat(op2.READ, coords.cell_node_map(),
                          flatten=True)]

       for c in coefficients:
           args.append(c.dat(op2.READ, c.cell_node_map(),
                             flatten=True))

       op2.par_loop(*args)
```

Inside par_loop

- Kernel optimised by COFFEE
    - LICM, padding and alignment, vectorisation
    - `arxiv:1407.0904 [cs.MS]`
    - For low-order extruded elements obtain 70% FP peak on prisms for matrix assembly.
- Wrapper code generated, compiled, loaded into process
- Call generated function with appropriate code

Boundary conditions

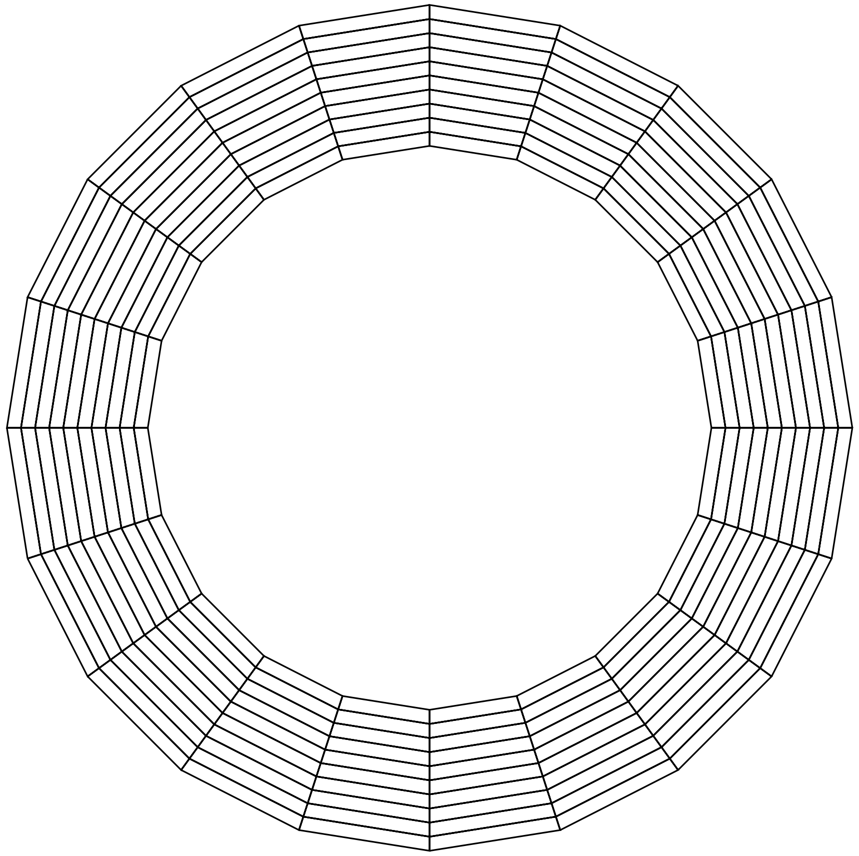
- http://firedrakeproject.org/boundary_conditions.html
- Effectively just throw away entries in Jacobian corresponding to bc dofs, put a 1 on the diagonal
  - PETSc makes this easy: pass negative row/column index to MatSetValues
- Set residual on boundary dofs to zero
- This approach maintains:
  - symmetry (or skew-symmetry)
  - positive (semi-)definiteness
  - diagonal dominance

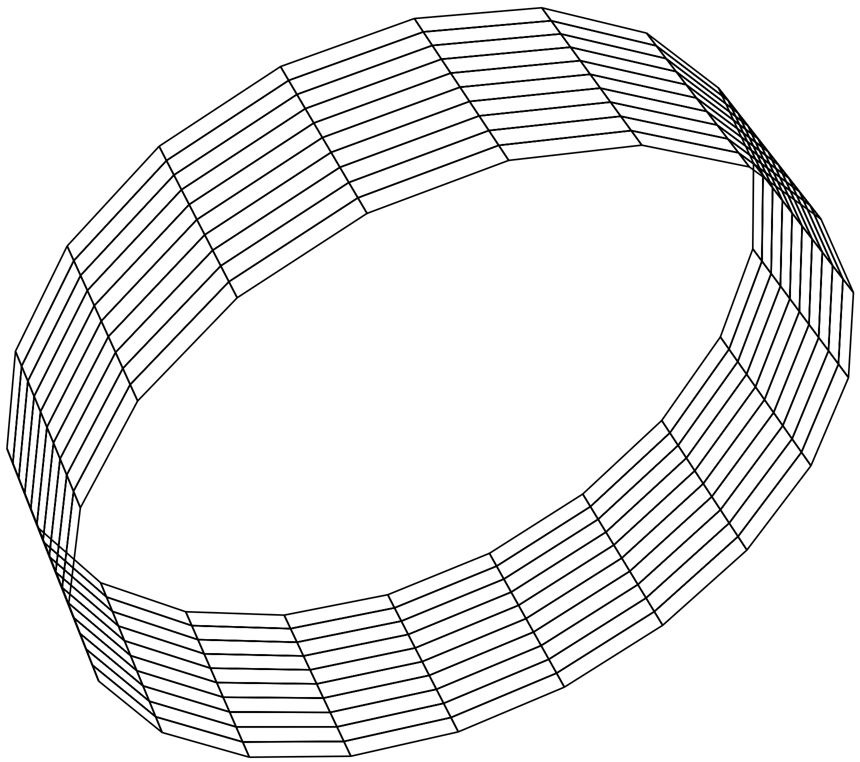
Extruded Firedrake

- *Extensions* to existing UFL syntax
- *Extensions* to PyOP2 types (seen these)
    - Not all extruded composes with non-extruded
    - can't do *direct* loops over extruded and non-extruded dats, can loop over extruded set and read non-extruded dat *indirectly*

Meshes

```python
from firedrake import *

m = CircleManifoldMesh(20, radius=10)
annulus = ExtrudedMesh(m, layers=10, layer_height=0.5,
                       extrusion_type='radial')
cylinder = ExtrudedMesh(m, layers=10, layer_height=0.5,
                        extrusion_type='uniform')

File('annulus.pvd') << annulus.coordinates
File('cylinder.pvd') << cylinder.coordinates
```

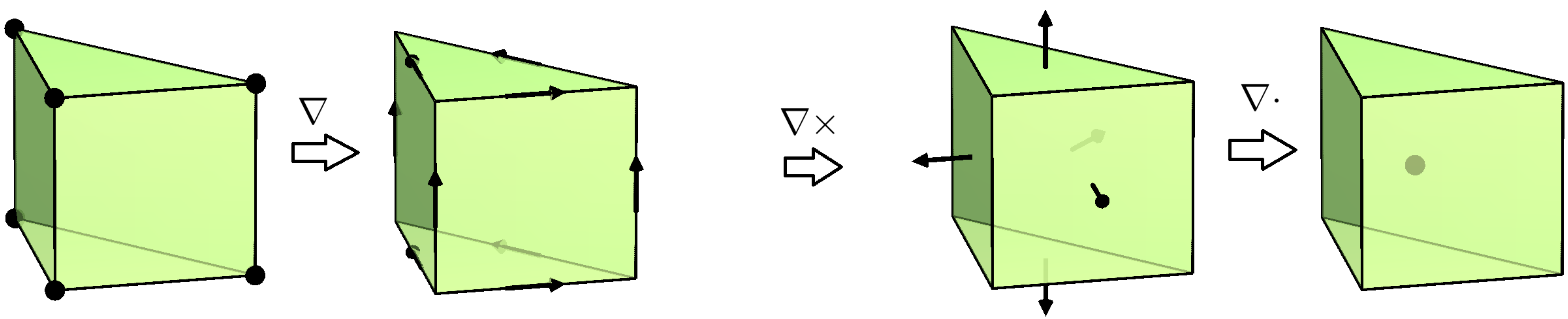




- *Uniform* extrusion increases topological and geometric dimension of cells
- *Radial* extrusion increases topological, but maintains geometric dimension
- Jacobians (currently) assume affine cells

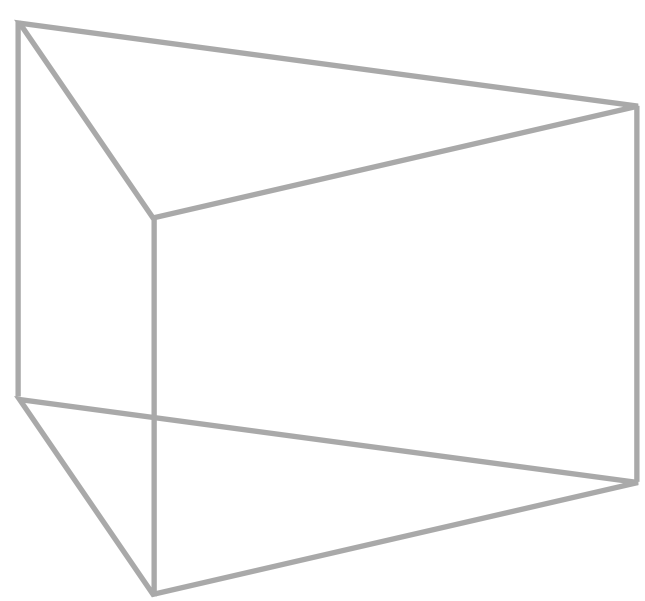
Elements

- Anything that can be expressed as tensor product of simplex (1 or 2d) with interval
  - e.g. Q2-P1dg for Stokes not supported
- You can have a full discrete de Rham product complex



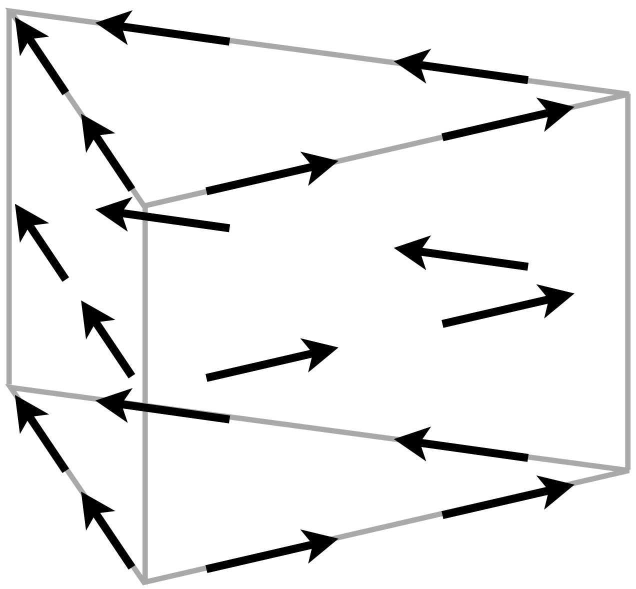
N2curl on prism

N2curl on prism



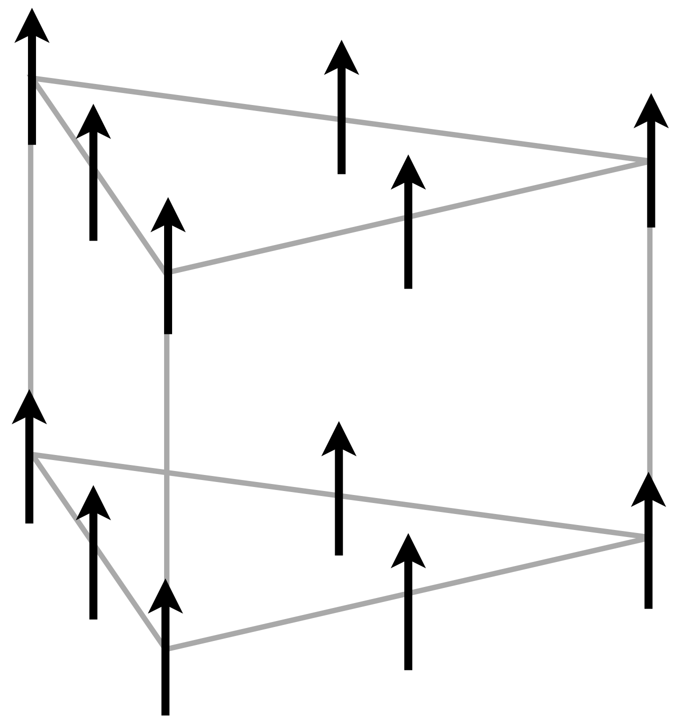
N2curl on prism

```
N2_1 = FiniteElement("N2curl", triangle, 1)
CG_2 = FiniteElement("CG", interval, 2)
Ned_horiz = HCurl(OuterProductElement(N2_1, CG_2))
```



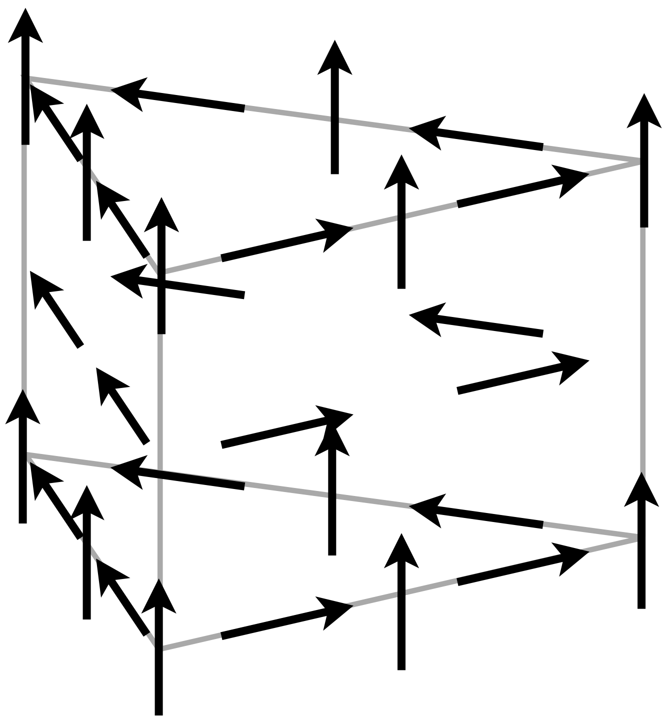
N2curl on prism

```
N2_1 = FiniteElement("N2curl", triangle, 1)
CG_2 = FiniteElement("CG", interval, 2)
Ned_horiz = HCurl(OuterProductElement(N2_1, CG_2))
P2tri = FiniteElement("CG", triangle, 2)
P1dg = FiniteElement("DG", interval, 1)
Ned_vert = HCurl(OuterProductElement(P2tri, P1dg))
```



N2curl on prism

```
N2_1 = FiniteElement("N2curl", triangle, 1)
CG_2 = FiniteElement("CG", interval, 2)
Ned_horiz = HCurl(OuterProductElement(N2_1, CG_2))
P2tri = FiniteElement("CG", triangle, 2)
P1dg = FiniteElement("DG", interval, 1)
Ned_vert = HCurl(OuterProductElement(P2tri, P1dg))
Ned_wedge = Ned_horiz + Ned_vert
```



Escaping abstractions

- What if I want to code a slope limiter?
  - UFL no good
- Firedrake allows you to code PyOP2 par_loops directly
  - Also some syntax to ease things
- PyOP2 doesn't have an "escape hatch"

- P1 field on vertices taking maximum value of P0 field on cells adjacent to those vertices

```
m = UnitSquareMesh(20,20)
cg = FunctionSpace(m, "CG", 1)
dg = FunctionSpace(m, "DG", 0)
c = Function(cg)
d = Function(dg)
par_loop("""for (int i=0; i<c.dofs; i++)
              c[i][0] = fmax(c[i][0], d[0][0]);""",
        # walk over cells
        dx,
        # associate 'c' variable with c field, 'd' with d
        {'c': (c, RW), 'd': (d, READ)})
```

Block solvers

- PETSc provides nice interface to block preconditioning
  - Schur complements (2x2 blocks)
  - Block Jacobi/Gauss-Seidel (nxn blocks)
- We (basically) inherit this
- Can either precondition with approximation of inverse of operator, or with approximation of inverse of some spectrally equivalent operator

Mixed Poisson

Find $\sigma \in \Sigma \subset \mathrm{H(div)}$, $v \in V \subset L^2$ satisfying

$$\begin{aligned} \langle \sigma, \tau \rangle - \langle u, \nabla \cdot \tau \rangle &= 0, \quad \forall\, \tau \in \Sigma, \\ \langle \nabla \cdot \sigma, v \rangle &= \langle f, v \rangle, \quad \forall\, v \in V \end{aligned}$$

A good preconditioner for this problem is the inverse of the H(div) inner product for the H(div) piece, and the inverse of the L^2 mass matrix for the L^2 piece
[Arnold, Falk, Winther, Multigrid in H(div) and H(curl)]

```python
from firedrake import *
m = UnitSquareMesh(40, 40)
Sigma = FunctionSpace(m, 'BDM', 2, name="sigma")
V = FunctionSpace(m, 'DG', 1, name="v")
W = Sigma * V
sigma, u = TrialFunctions(W)
tau, v = TestFunctions(W)
n = FacetNormal(m)
a = (inner(sigma, tau) - div(tau)*u + div(sigma)*v)*dx
L = - 4*dot(tau, n)*ds(4) - 2*dot(tau, n)*ds(3)
aP = (inner(sigma, tau) + div(sigma)*div(tau) + u*v)*dx
bcs = [DirichletBC(W[0], (0, 0), (1, 2))]
w = Function(W)
solve(a == L, w, Jp=aP, bcs=bcs,
      solver_parameters={'ksp_type': 'gmres',
                         'pc_type': 'fieldsplit',
                         'ksp_converged_reason': True,
                         'pc_fieldsplit_type': 'additive',
                         'fieldsplit_sigma_ksp_type': 'preonly',
                         'fieldsplit_sigma_pc_type': 'lu',
                         'fieldsplit_v_ksp_type': 'richardson',
                         'fieldsplit_v_pc_type': 'jacobi',
                         'fieldsplit_v_ksp_max_it': 5})

uexact = Function(V)
uexact.interpolate(Expression("2 + 2*x[1]"))
sigma, u = w.split()
assert errornorm(u, uexact) < 1e-6
```

Could have used full Schur complement PC on original system, converges in fewer iterations but takes about 5x as long.

```
solve(a == L, w, bcs=bcs,
     solver_parameters={'ksp_type': 'gmres',
                        'pc_type': 'fieldsplit',
                        'ksp_converged_reason': True,
                        'ksp_monitor_true_residual': True,
                        'pc_fieldsplit_type': 'schur',
                        'pc_fieldsplit_schur_fact_type': 'full',
                        'fieldsplit_sigma_ksp_type': 'preonly',
                        'fieldsplit_sigma_pc_type': 'lu'})
```

Obviously for large problems we can't invert with LU, working on good PCs for problems in H(div) and H(curl)

More questions?