Firedrake

pyop3: A new domain-specific language for automating high-performance mesh-based simulation codes

Connor Ward (supervised by David Ham)

January 2023

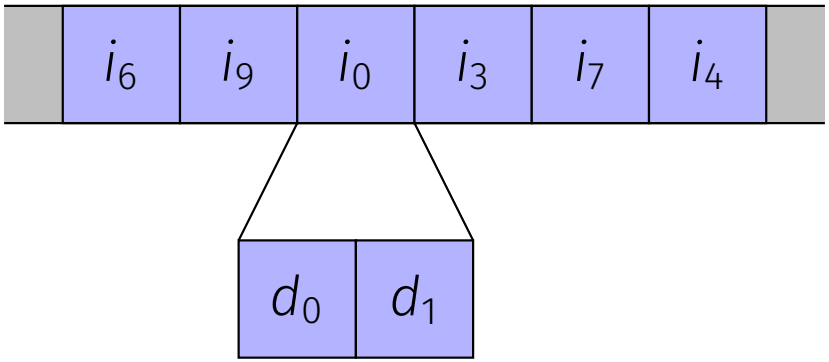
Imperial College London

What came before: PyOP2

- Domain-specific language embedded in Python for doing mesh computations
- Uses code generation to produce fast code
- Handles the data structures used by Firedrake
- Used everywhere in Firedrake for things like residual assembly and interpolation

PyOP2 data model

- Data is stored by `Dats`[1]
- These associate a fixed inner shape (d_m) with a set of possibly unordered nodes (i_n)
- `Mixed Dats` and `Dats` for extruded meshes are also possible



[1]Sparse matrix support is not discussed

Introducing pyop3

Key differences with PyOP2:

- Complete rewrite of the code generation part
- New more expressive and composable interface inspired by PETSc DMPlex
- **Has a new data layout model**

pyop3 wishlist

Features:

- ☐ Handle orientations (e.g. unstructured hexes)
- ☐ p-adaptivity
- ☐ Mixed meshes

Performance:

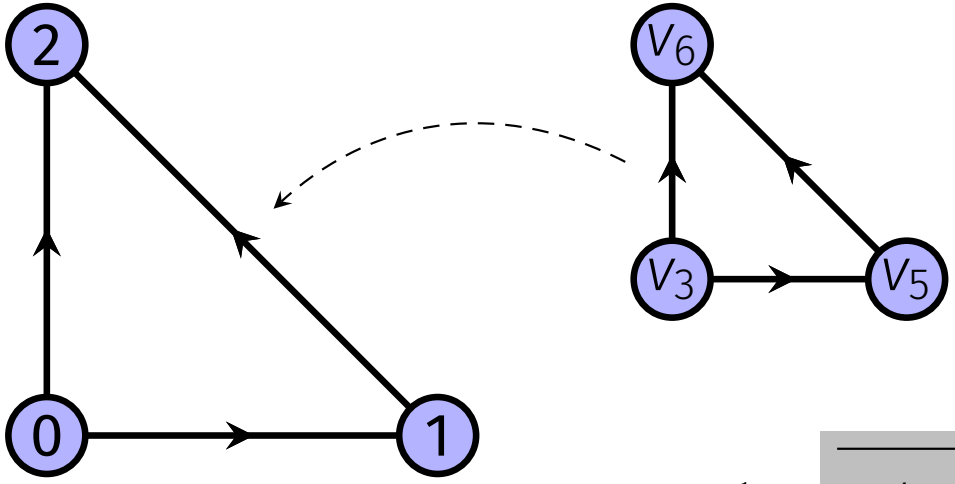
- ☐ Exploit mesh partial structure (e.g. extruded)[2]
- ☐ Prescribe DoF ordering (e.g. extruded columns)[2]

(And more...)

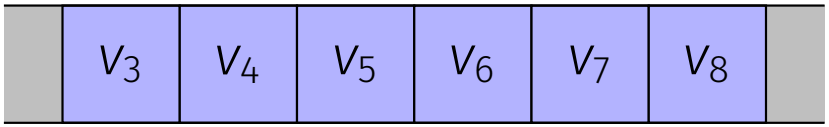
[2] Achievable in PyOP2 but difficult to use

Claim: pyop3's new data layout abstraction enables all of these.

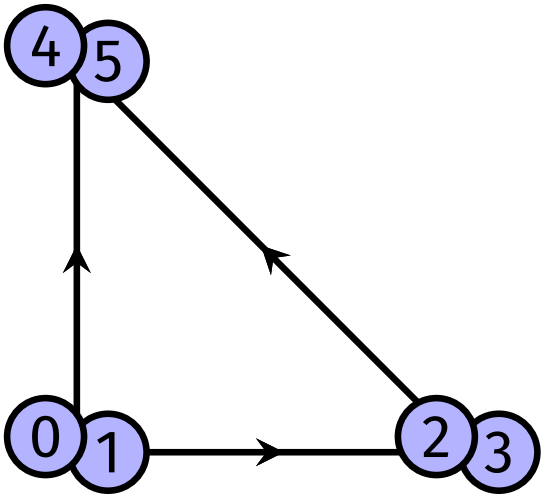
Starting simple: P1

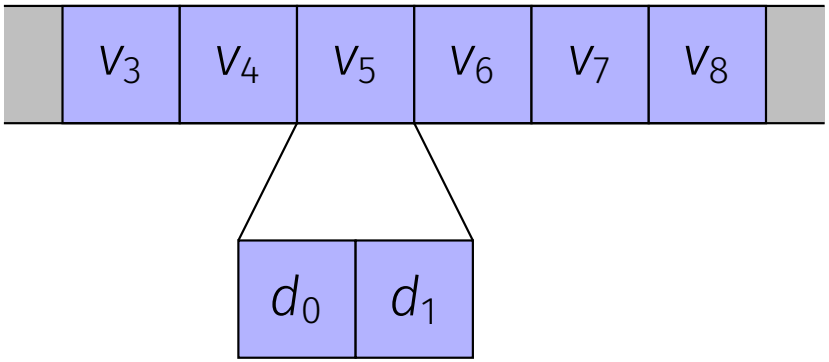



```
1  root = (
2    MultiAxis()
3    .add_part(AxisPart(nverts))
4  )
```



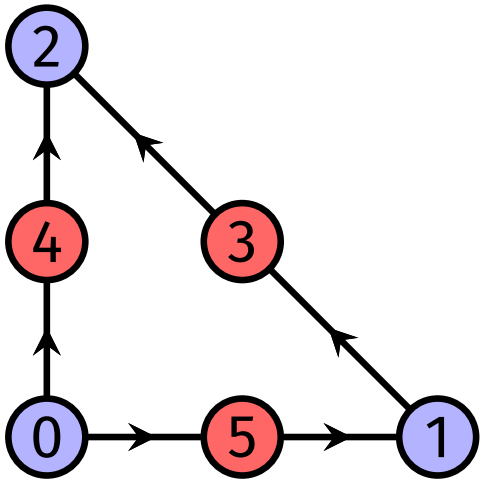
Adding shape: vector P1

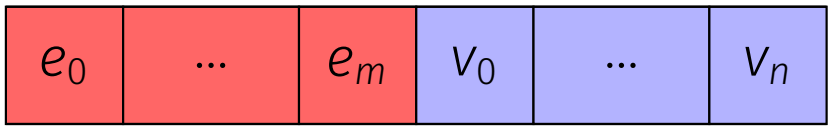





```
root = (
  MultiAxis()
  .add_part(AxisPart(nverts))
  .add_subaxis(AxisPart(2))
)
```

Multiple entities: P2



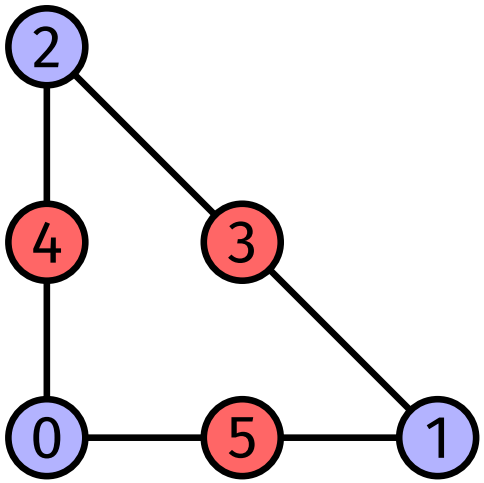


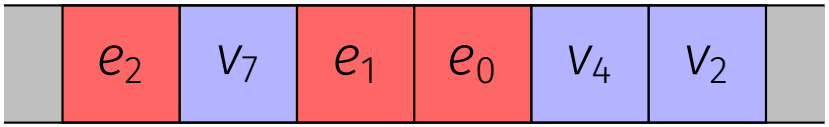
- ☑ p-adaptivity[3]
- ☑ Mixed meshes[3]

```
1  root = (
2    MultiAxis()
3    .add_part(AxisPart(nedges))
4    .add_part(AxisPart(nverts))
5  )
```

[3] Since topological entities are now distinguishable

Now with renumbering

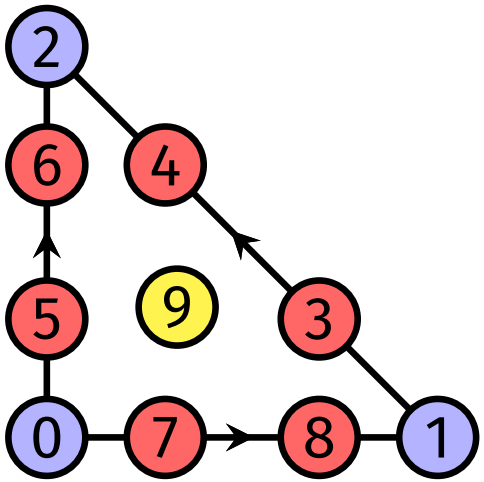


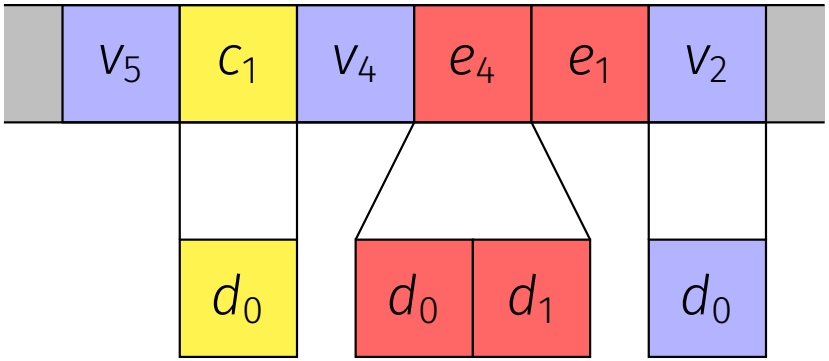


☑ Prescribe DoF ordering

```
root = (
  MultiAxis()
  .add_part(AxisPart(
    nedges,
    numbering=[4,2,5,...],
  ))
  .add_part(AxisPart(
    nverts,
    numbering=[3,0,1,...],
  ))
)
```

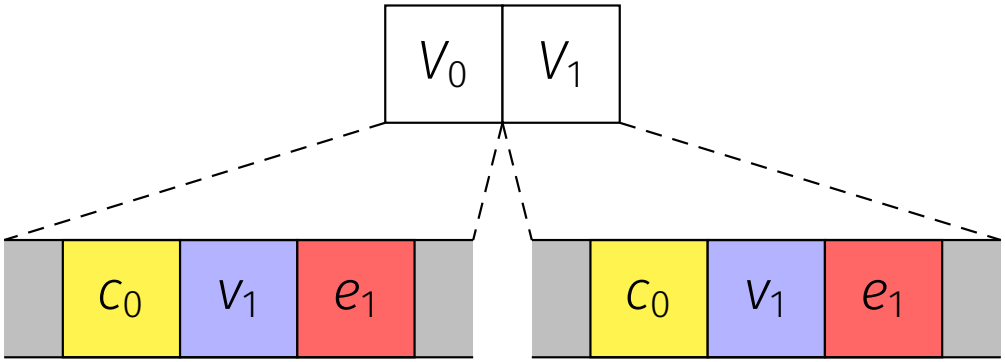
More complicated inner shape: P3






```
1  root = (
2    MultiAxis()
3    .add_part(AxisPart(ncells, "cells"))
4    .add_part(AxisPart(nedges, "edges"))
5    .add_part(AxisPart(nverts, "verts"))
6    .add_subaxis("edges", AxisPart(2))
7  )
```

Mixed




```
root = (
  MultiAxis()
   .add_part(AxisPart(1, "V0"))
   .add_part(AxisPart(1, "V1"))
   .add_subaxis("V0", ...)
   .add_subaxis("V1", ...)
)
```

pyop3 wishlist

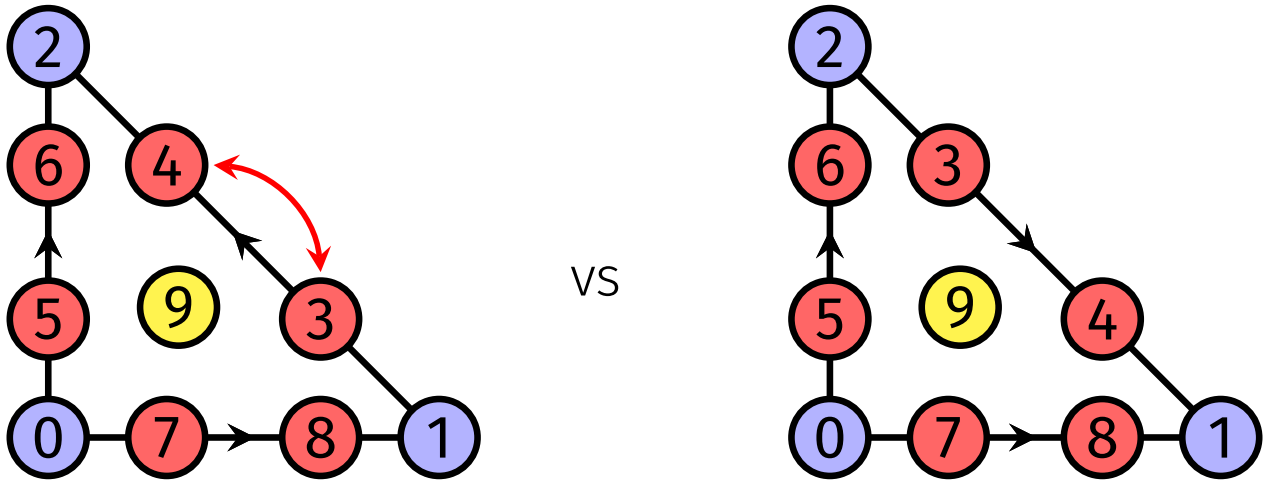
Features:

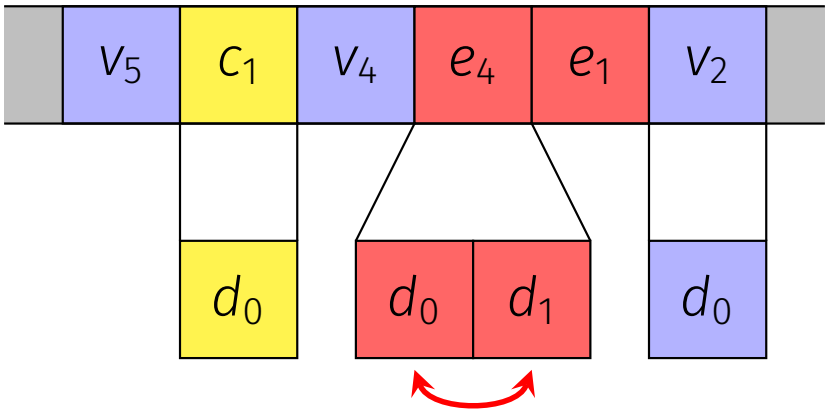
- ☐ Handle orientations
- ☑ p-adaptivity
- ☑ Mixed meshes

Performance:

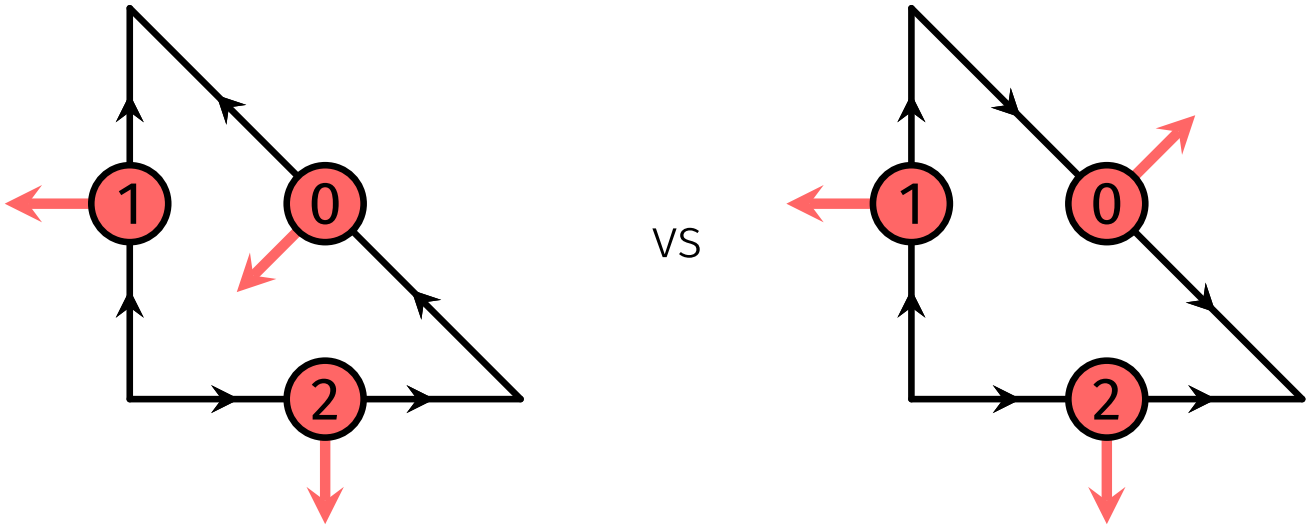
- ☐ Exploit mesh partial structure
- ☑ Prescribe DoF ordering

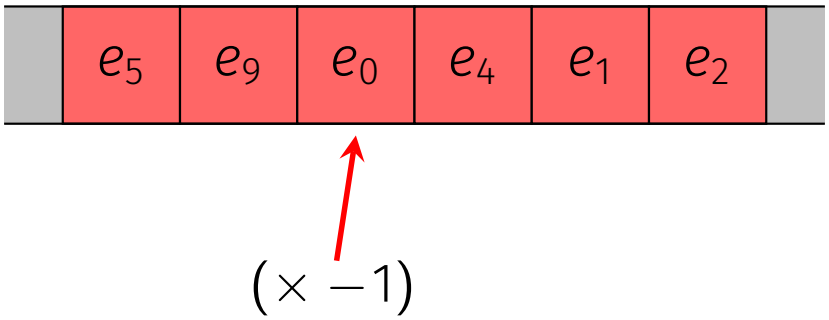
Orientation: P3



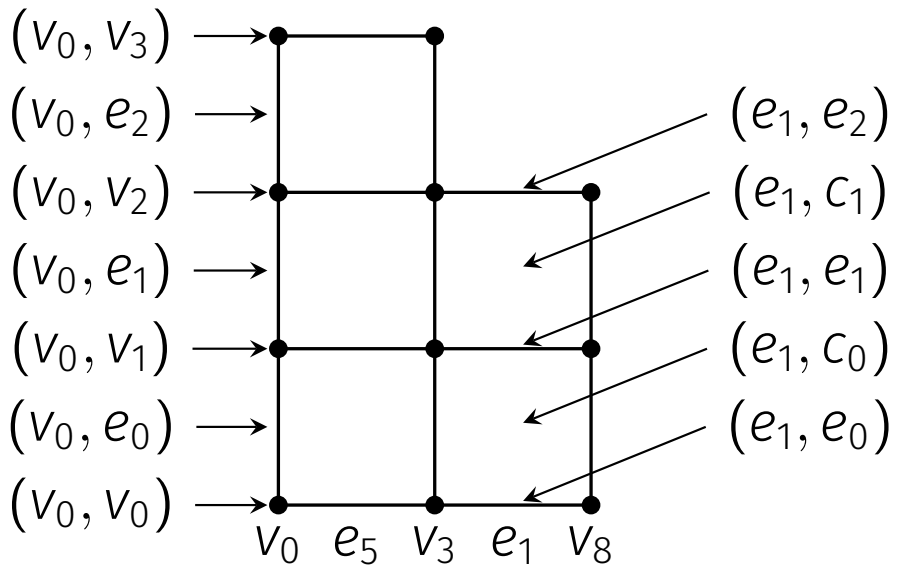


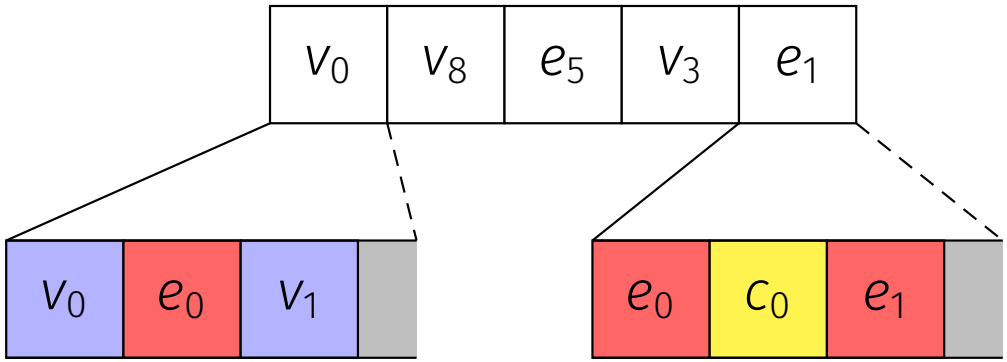
Orientation: Raviart-Thomas





Partially-structured meshes: extruded





pyop3 wishlist

Features:

☑ Handle orientations
☑ p-adaptivity
☑ Mixed meshes

Performance:

☑ Exploit mesh partial structure
☑ Prescribe DoF ordering

Things I missed

- The interface for launching computations (inc. map and loop composition)
- Tight integration with PETSc (esp. DMPlex)
- Support for sparse matrices
- MPI parallelism
- Could streamline PCPATCH and multigrid code (via composition)
- Should retain PyOP2's work on GPUs and inter-element vectorisation
- Additional data layout transformations/optimisations
- Could potentially do a similar mesh structure trick for refined meshes

Appendix

pyop3 interface

```
do_loop(
  c := mesh.cells.index,
  kernel(dat0[closure(c)], dat1[closure(c)])
)

do_loop(
  f := mesh.interior_facets.index,
  kernel(
    dat0[closure(support(f))],
    dat1[closure(support(f))]
  )
)
```

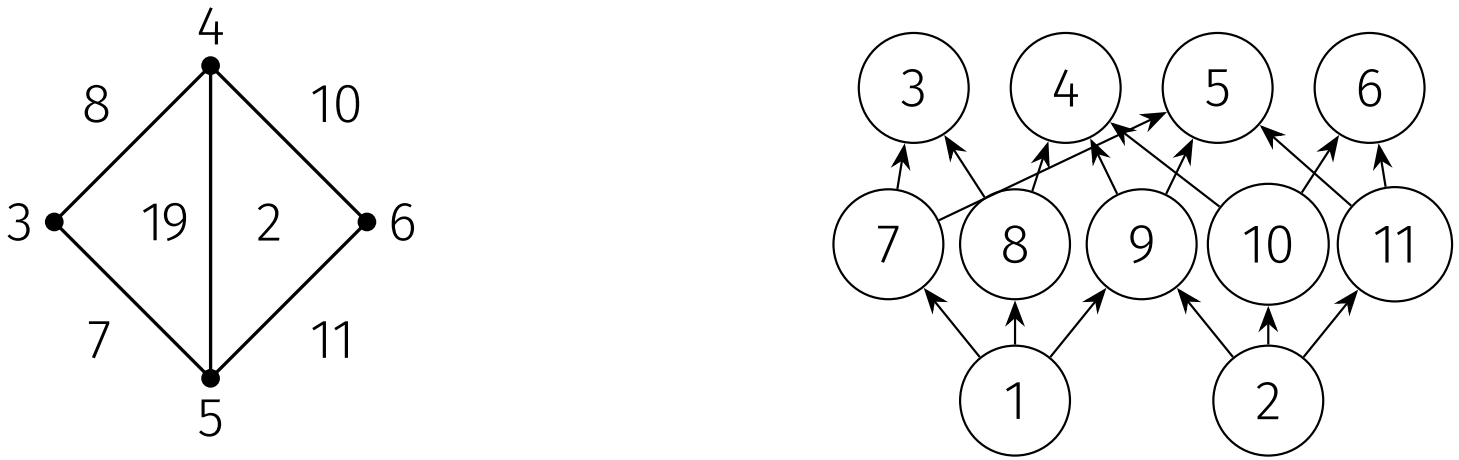
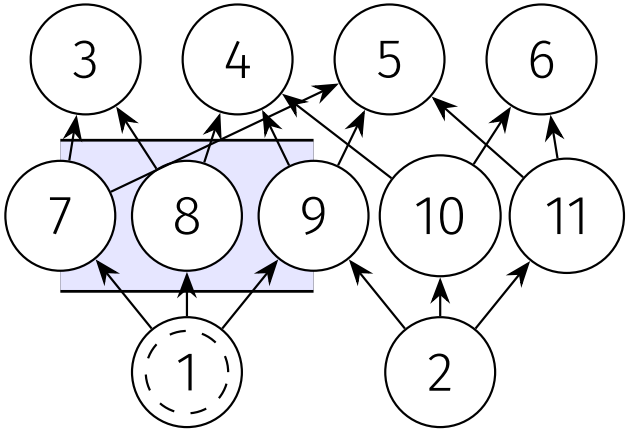
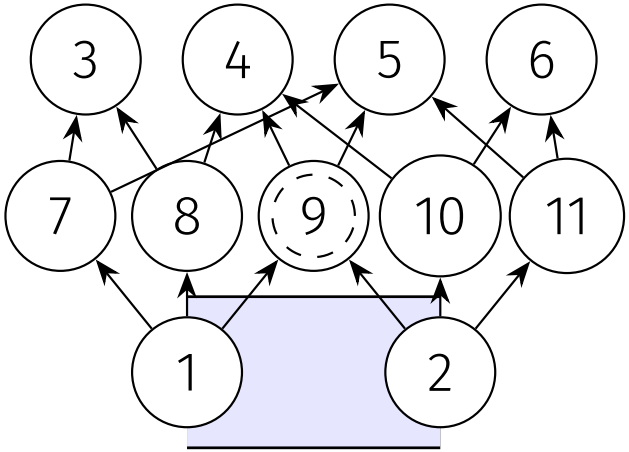


Figure 1: An example mesh and its Hasse diagram representation. Note that the topological entities are numbered according to the DMPlex convention of first cells, then vertices, then faces.

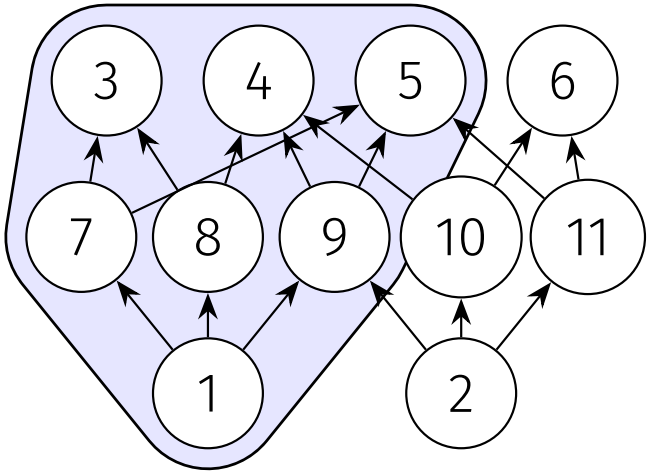
DMPlex 2



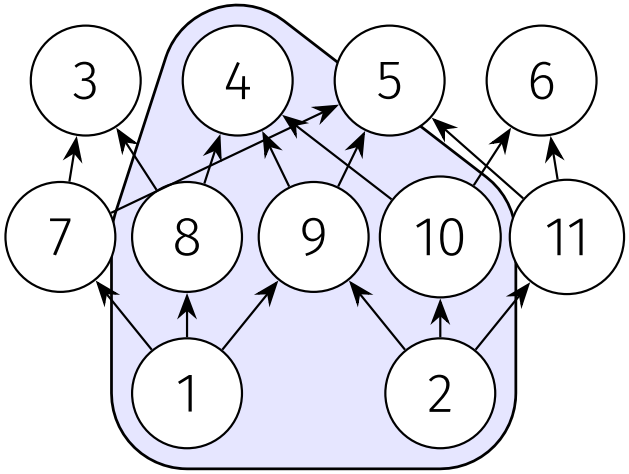
(a) $\mathbf{cone}(1) = \{7, 8, 9\}$



(b) $\mathbf{supp}(9) = \{1, 2\}$



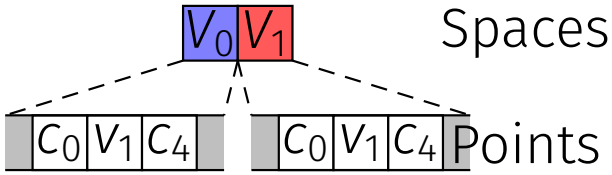
(c) $\mathbf{cl}(1) = \{1, 3, 4, 5, 7, 8, 9\}$



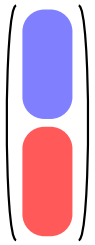
(d) $\mathbf{st}(4) = \{4, 8, 9, 10, 1, 2\}$

Figure 2: The possible DMPlex covering queries (applied to the Hasse

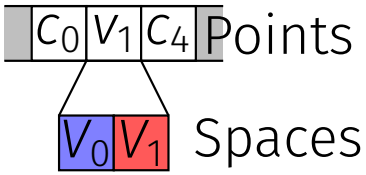
Mixed reordering



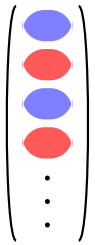
(a) A typical data layout for a 'mixed' system with the spaces V_0 and V_1 forming the 'outer' axis.



(b) The resulting block-structured vector.



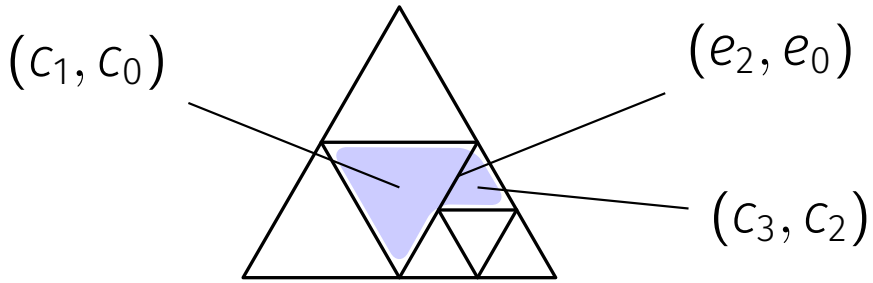
(c) A transformed data layout where the "Spaces" and "Points" axes have been swapped.



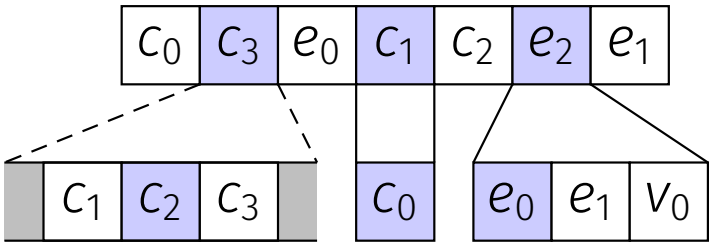
(d) The resulting interleaved vector.

Figure 3: A possible data layout transformation for a 'mixed' system permitted by **pyop3**. The entries V_0 and V_1 represent the spaces of the mixed system and the "Points" axis is representative of the mesh

Partially-structured meshes: refined



(a) An example of a stencil - $\mathfrak{st}((e_2, e_0))$ - over a refined mesh. Note that the unrefined cell (c_1, c_0) is still indexed with two indices. We say that it has been refined using the identity transformation.



(b) Example data layout for the refined mesh shown above. Note that the base mesh in unstructured which is why the top axis is unordered.

Figure 4: Example data layout and stencil for a refined mesh.

PCPATCH

```
loop(v := mesh.vertices.index, [
  loop(p := star(v).index, [
    assemble_jacobian(dat1[closure(p)], dat2[closure(p)], "mat"),
    assemble_residual(dat3[closure(p)], "vec"),
  ]),
  solve_and_update("mat", "vec", dat4[v]),
])
```