Latest developments in pyop3

Connor Ward

13 September 2023

Overview

What is **pyop3**?

A simple-ish example

What's the point?

Imperial College London

What is `pyop3`?

TL;DR

- A programming language for mathematicians
- Comes with a compiler
- The language lets you express how to read and write from complicated data structures

In more detail

- A **domain-specific** programming language for mathematicians **embedded in Python (like UFL)**
- Comes with a **just-in-time** compiler **that targets loopy and then C/CUDA/OpenCL**
- The language lets you express how to read and write from complicated data structures
- **Never need to create a `PetscSection` ever again!**

Why is this hard?

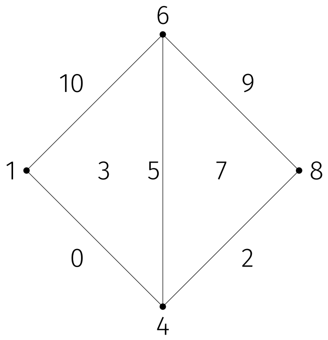
- FEM codes have diverse and complicated data structures
- These data structures also need to be accessed in non-trivial ways

A simple-ish example

Outline of a pyop3 program

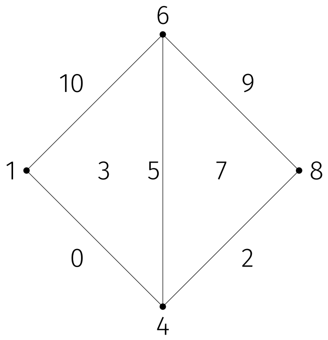
1. Create data structures
2. Execute loop expressions acting on the data structures

Create a data layout for a 2 cell mesh




```
mesh_axis = Axis(
  [
    AxisComponent(2, "cells"),
    AxisComponent(5, "edges"),
    AxisComponent(4, "verts"),
  ],
  "mesh"
  permutation=[3, 7, 0, 10, 5, 9, 2, 1, 6, 4, 8],
)
```

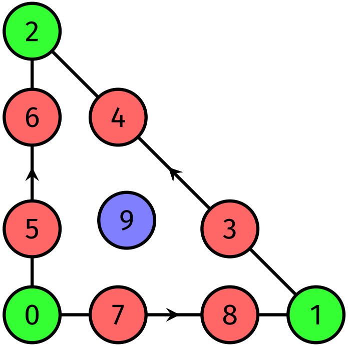
Create a data layout for a 2 cell mesh




```
mesh_axis = Axis(
  [
    AxisComponent(2, "cells"),
    AxisComponent(5, "edges"),
    AxisComponent(4, "verts"),
  ],
  "mesh"
  permutation=[3, 7, 0, 10, 5, 9, 2, 1, 6, 4, 8],
)
```

0	0	4	0	2	2	1	1	3	3	1

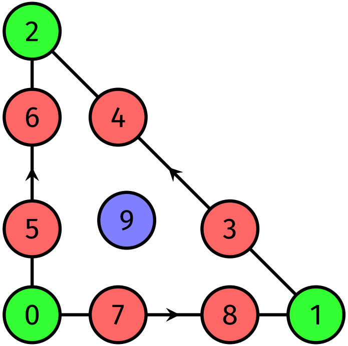
Now make it a P3 function space



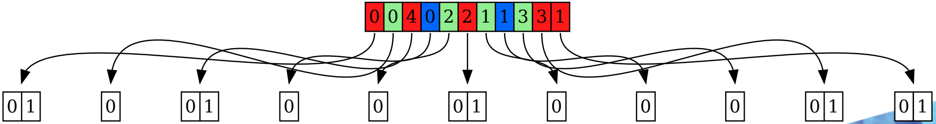

```
axes = (
  AxisTree(mesh_axis)
    .add_subaxis(Axis(1), mesh_axis.id, "cells")
    .add_subaxis(Axis(2), mesh_axis.id, "edges")
    .add_subaxis(Axis(1), mesh_axis.id, "verts")
)
```

Imperial College London

Now make it a P3 function space

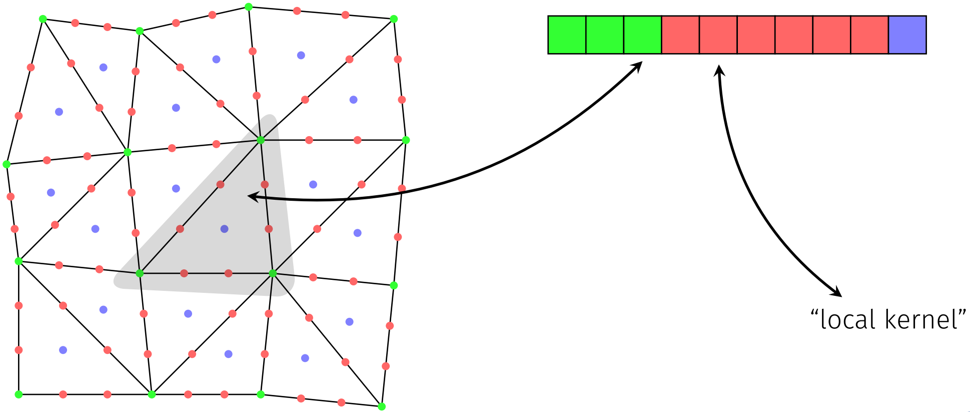



```
axes = (
  AxisTree(mesh_axis)
    .add_subaxis(Axis(1), mesh_axis.id, "cells")
    .add_subaxis(Axis(2), mesh_axis.id, "edges")
    .add_subaxis(Axis(1), mesh_axis.id, "verts")
)
```



Imperial College London

Now let's do a loop



Residual assembly in pyop3

```
for every cell in the mesh:
  collect DoFs found in the cell's closure
  call a local kernel with these DoFs
  scatter the result to a global vector
```

Residual assembly in pyop3

```
for every cell in the mesh:
  collect DoFs found in the cell's closure
  call a local kernel with these DoFs
  scatter the result to a global vector
```

```
loop(
  c := mesh.cells.index(),
  kernel(func0[closure(c)], ...)
)
```

Imperial College London

Code generation!

```
void my_loop(double *func0, int *map0, int *map1, int *layout0, int *layout1, int *layout2, ...) {
  double t_0[10];  // to store the "packed" data

  for (int32_t i_0 = 0; i_0 < 2; ++i_0) {                     // loop over cells
    // pack cell DoFs
    t_0[0] = func0[layout0[i_0]];

    // pack edge DoFs
    for (int32_t i_5 = 0; i_5 < 3; ++i_5) {                   // loop over edges
      for (int32_t i_6 = 0; i_6 < 2; ++i_6) {                 // loop over edge DoFs
        j_3 = map0[i_0 * 3 + i_5];                            // select the right edge
        t_0[i_5*2 + i_6 + 1] = func0[layout1[j_3] + i_6];   // pack DoF
      }
    }

    // pack vertex DoFs
    for (int32_t i_7 = 0; i_7 < 3; ++i_7) {                   // loop over vertices
      j_5 = map1[i_0 * 3 + i_7];                              // select the right vertex
      t_0[i_7 + 7] = func0[layout2[j_5]];                     // pack DoF
    }

    // execute the local kernel and unpack the result
    kernel(t_0, ...);
    ...
  }
}
```

What’s the point?

Imperial College London

But PyOP2 can already do this, why do we need pyop3?

Is it faster than PyOP2?

But PyOP2 can already do this, why do we need **pyop3**?

Is it faster than PyOP2?

No!

But PyOP2 can already do this, why do we need **pyop3**?

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

But PyOP2 can already do this, why do we need pyop3?

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

Not yet!

But PyOP2 can already do this, why do we need pyop3?

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

Not yet!

So why is it useful?

But PyOP2 can already do this, why do we need pyop3?

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

Not yet!

So why is it useful?

It’s not.

But PyOP2 can already do this, why do we need **pyop3**?

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

Not yet!

So why is it useful?

~~It's not.~~

Is it faster than PyOP2?

No!

Is it as fast as PyOP2?

Not yet!

So why is it useful?

~~It's not.~~

Composability!

Why is composability important?

- Performance optimisation is not the main priority, expressibility is
- PyOP2 has limitations:
  - Single loop
  - Single kernel
  - No map composition
  - Extruded meshes require invasive code changes
- Can do more in fewer lines of code
  - `pyop3` compiler is $\sim$ 1000 lines of code, PyOP2's is $\sim$ 2500
  - No special casing for extruded

Imperial College London

Map composition

- Interior facet integrals:

```
loop(facet := mesh.interior_facets, kernel(func0[closure(support(facet))]))
```

- Multigrid:

```
closure(fine2coarse(fine_cell))
```

Imperial College London

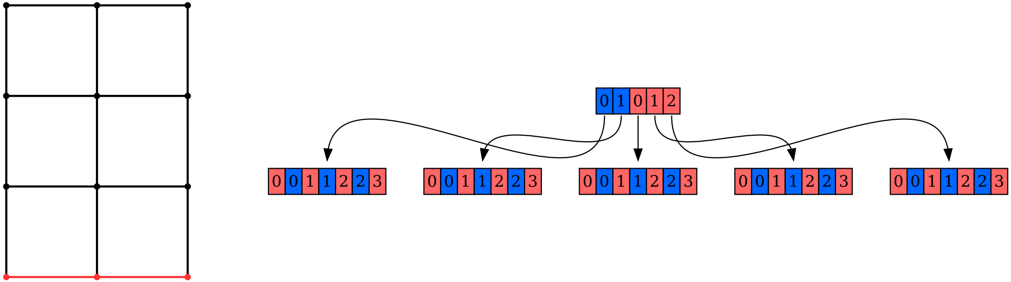
Loop and kernel composition: PCPATCH

```
loop(v := mesh.vertices.index, [
  loop(c := star(v).index, [
    kernel1(dat1[closure(c)], "mat"),
    kernel2(dat2[closure(c)], "vec")
  ]),
  solve("mat", "vec", dat3[v])
])
```

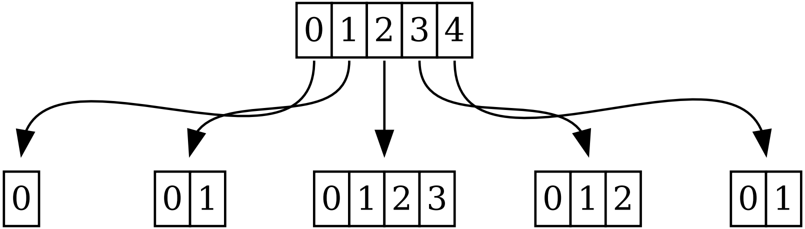
- We could also try to rewrite SLATE
- Loop composition can enable certain tiling optimisations

And this composability will work with LOADS of data structures...

Example 1: extruded



Example 2: ragged

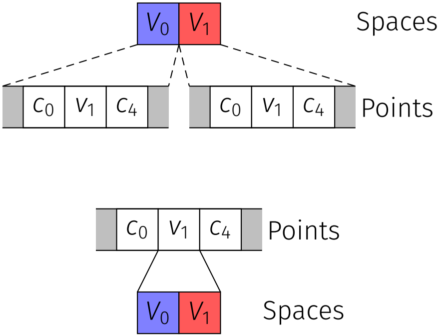


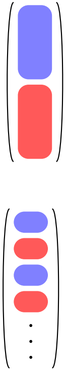
Useful for variable-layer extrusion and PIC.

Example 3: sparse

- Like ragged (no picture sorry)
- Arbitrary sparsity is completely possible

Example 4: “swapping” axes





Summary

- pyop3 is a DSL/compiler framework for writing kernels with non-trivial access patterns
- It can do everything PyOP2 can do, and more!
- WIP

Imperial College London