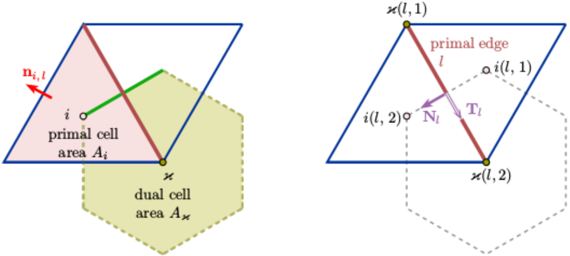
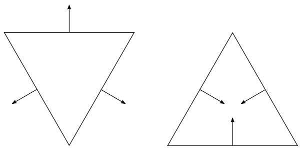
Recap



$$\mathrm{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} (N_l \cdot n_{i,l}) l$$

Recap



$$\mathrm{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} l \quad \text{for downward cells}$$

$$\mathrm{div}(v)_i = -\frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} l \quad \text{for upward cells}$$

```
auto ff = [](const double _in1, const double _in2, const
    double _res) -> double
    { return _in1 * _in2 + _res; };
eval(out_cells()) =
    eval(on_edges(ff, 0.0, in_edges(), edge_length())) /
        eval(cell_area());
```

From

$$\operatorname{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} l$$

to

$$\operatorname{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} (N_l \cdot n_{i,l}) l = \sum_{l \in \mathcal{E}(i)} w_{i,l} v_{n_l}$$

- One multiplication less. (Not necessarily faster)
- More general. (Definitely needed)

More operators

$$\mathrm{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} (N_l \cdot n_{i,l}) l = \sum_{l \in \mathcal{E}(i)} w_{i,l} v_{n_l}$$

edge-to-cell averaging:

$$\bar{\varrho}_i = \sum_{l \in \mathcal{E}(i)} \frac{A_{i,l}}{A_i} \varrho_l = \sum_{l \in \mathcal{E}(i)} w_{i,l} v_{n_l}$$

cell-to-edge averaging:

$$\breve{\varrho}_l = \sum_{j=1,2} \frac{A_{i(l,j)}}{A_l} \varrho_{i(l,j)} = \sum_{j=1,2} w_{i(l,j)} \varrho_{i(l,j)}$$

Used in discretizing the pressure gradient force:

$$\overline{\overline{\psi}}_l = \frac{A_{i(l,2),l}}{A_l} \psi_{i(l,1)} + \frac{A_{i(l,1),l}}{A_l} \psi_{i(l,2)} = \sum_{j=1,2} w_{i(l,j)} \psi_{i(l,j)}$$

Fields on multiple locations

{i,c,j,k}
{i,c,j,k,extra_dimension} (new!)
How to access:

```
edges_of_cells_storage_type weights;
weights(i, c, j, k, 2);
```

In a functor:

```
typedef in_accessor<1, icosahedral_topology_t::cells,
    extent<1>, 5 > weights;

using edge_of_cells_dim = dimension< 5 >;
edge_of_cells_dim::Index edge;
eval(weights(edge+2));
```

Other than multiple-location `weights`

What else do we need?

$$\mathrm{div}(v)_i = \sum_{l \in \mathcal{E}(i)} w_{i,l} v_{n_l}$$

`on_edges()` are currently not dealing with multiple-location fields. We need to fold manually. But when we use `eval(weights(edge+0))` to get $w_{i,0}$, how do we get v_{n_0}?

→ expose topology to user

→ fix neighbors order

Expose topology to user

```
/*
 * this stencil is primarily on cells
 */

typedef in_accessor<0, icosahedral_topology_t::edges,
    extent<1> > in_edges;

auto neighbors_offsets = connectivity< cells , edges
    >::offsets(eval.position()[1]);

// get the first edge on cell
eval(in_edges(neighbor_offsets[0]));

// iterate over all edges of cell
for (auto neighbor_offset : neighbors_offsets)
  eval(in_edges(neighbor_offset));
```

Fix neighbors order

```
Neighbor edges of a cell must follow the same
    convention than neighbor cells of a cell.
I.e. the following
          /\
         1  2
        /_0__\
 imposes
      ____________
      \    /\    /
       \1 /  \2 /
        \/____\/
         \  0 /
          \  /
           \/
```

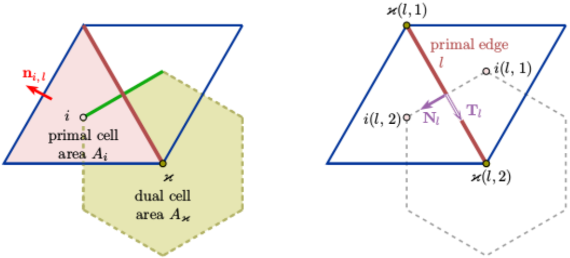
Now `eval(weights(edge+0))` and `eval(in_edges(neighbor_offsets[0]))` point to the same direction.

Combining everything, now a div

```
template<typename Evaluation>
GT_FUNCTION static void Do(Evaluation const &eval,
    x_interval)
{
    typedef typename icgrid::get_grid_topology< Evaluation
        >::type grid_topology_t;

    // for multiple-location fields
    using edge_of_cells_dim = dimension< 5 >;
    edge_of_cells_dim::Index edge;

    // for exposed topology
    auto neighbors_offsets = connectivity< cells , edges >::
        offsets(eval.position()[1]);

    eval(out_cells()) = 0.;
    ushort_t e=0;
    for (auto neighbor_offset : neighbors_offsets) {
        eval(out_cells()) += eval(in_edges(neighbor_offset))
             * eval(weights(edge+e));
        e++;
    }
}
```

We will discuss how to make this look better later.

grad



$$\mathrm{grad}_n(\varrho)_l = \frac{\varrho_{i(l,2)} - \varrho_{i(l,1)}}{\hat{l}}$$

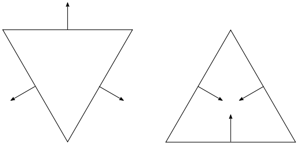
$\rightarrow$ fix neighbour order

A grad

$$\text{grad}_n(\varrho)_l = \frac{\varrho_{i(l,2)} - \varrho_{i(l,1)}}{\hat{l}}$$

```
typedef in_accessor<0, icosahedral_topology_t::cells, extent
    <1> > in_cells;
typedef in_accessor<1, icosahedral_topology_t::edges, extent
    <1> > dual_edge_length;
typedef inout_accessor<2, icosahedral_topology_t::edges>
    out_edges;

template<typename Evaluation>
GT_FUNCTION static void Do(Evaluation const &eval,
    x_interval) {
    auto neighbors_offsets =
      connectivity<edges, cells>::offsets(eval.position()
          [1]);

    eval(out_edges()) =
      ( eval(in_cells(neighbors_offsets[0])) -
        eval(in_cells(neighbors_offsets[1]))
      ) / eval(dual_edge_length());
}
```

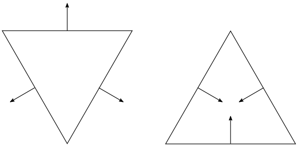
Neighbors order



Cell neighbours of an edge, in the order $0 \rightarrow 1$ follow the direction of the N_l.

```
neighbors order
                  /\
                 / 0\
      ______             ______
     /\  1 /    /____\   \ 1  /\
    /0 \  /     \    /    \  / 0\
   /____\/       \1 /      \/____\
                  \/
```

Neighbors order



Vertex neighbors of an edge, in the order $0 \rightarrow 1$ defines a vector N_l which is perpendicular to N_t.

```
neighbors order
    1______         /\          ______0
    /\     /       /  \         \     /\
   /  \   /      0/____\1        \   /  \
  /____\/         \    /          \/____\
        0          \  /           1
                    \/
```

Useful for

$$\mathrm{grad}_\tau(\varrho)_l = \frac{\varrho_{\varkappa(l,2)} - \varrho_{\varkappa(l,1)}}{l}$$

Neighbors order

We also set rules for neighbors of vertexes. Helpful for a curl.

Evaluation

$$\mathrm{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} (N_l \cdot n_{i,l}) l$$

- weights: $\mathrm{div}(v)_i = \sum_{l \in \mathcal{E}(i)} w_{i,l} v_{n_l}$
- flow convention: $\mathrm{div}(v)_i = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l} l$
- over edges:

$$\mathrm{div}(v)_{\varkappa(l,j)} \leftarrow \mathrm{div}(v)_{\varkappa(l,j)} \pm v_{n_l} l \quad j = 1, 2 \tag{1}$$

$$\mathrm{div}(v)_{\varkappa(l,j)} \leftarrow \mathrm{div}(v)_{\varkappa(l,j)} / A_{\varkappa(l,j)} \quad j = 1, 2 \tag{2}$$

Approach	Primal location	Operation on primal location
weights	Cells	6 read, 1 write, 3 mult, 3 plus
flow convention	Cells	7 read, 1 write, 4 mult, 3 plus
over edges	Edges	4 read, 2 write, 1 mult, 2 plus
		+ 4 read, 2 write, 2 mult

$$n_{cell} : n_{edge} = 2 : 3$$

Evaluation

	CPU (10 times)			GPU (100 times)		
Approach	R02B05	R02B06	R02B07	R02B05	R02B06	R02B07
weights	0.06777	0.3174	1.247	0.03529	0.07045	0.2834
flow convention	0.08640	0.4236	1.631	0.02927	0.06547	0.3110
over edges	0.1010	0.4821	1.907	0.05403	0.1015	0.3934

Table: performance numbers on kesch, one patch of the global grid, klevel=50

Syntactic sugar

```
// on_edges
auto ff = [](const double _in1, const double _in2, const
    double _res) -> double { return _in1 * _in2 + _res; };
eval(out_cells()) = eval(on_edges(ff, 0.0, in_edges(),
    edge_length()));

// but with five dimension fields
using edge_of_cells_dim = dimension< 5 >;
edge_of_cells_dim::Index edge;
auto neighbors_offsets = connectivity< cells , edges >::
    offsets(eval.position()[1]);

eval(out_cells()) = 0.;
ushort_t e=0;
for (auto neighbor_offset : neighbors_offsets) {
    eval(out_cells()) += eval(in_edges(neighbor_offset)) *
        eval(weights(edge+e));
    e++;
}

// extend on_edges
using edge_of_cells_dim = dimension< 5 >;
auto ff = ...;
eval(out_cells()) = eval(on_edges(ff, 0.0, in_edges(),
    weights(edge_of_cells_dim)));
```

Syntactic sugar

```
// grad
auto neighbors_offsets =
  connectivity<edges, cells>::offsets(eval.position()[1]);

eval(out_edges()) =
  ( eval(in_cells(neighbors_offsets[0])) -
    eval(in_cells(neighbors_offsets[1]))
  ) / eval(dual_edge_length());

// proposal
eval(out_edges()) = eval(on_cells(reduction_functor, 0.0,
    in_cells(), {1,  -1})) / eval(dual_edge_length());
```

Syntactic sugar

But topology exposure is still necessary:

$$\mathrm{div}(v)_{\varkappa(l,j)}\pm = v_{n_l} l \quad j = 1, 2$$

```
double t{eval(in_edges()) * eval(edge_length())};
eval(out_cells(neighbors_offsets[0])) -= t;
eval(out_cells(neighbors_offsets[1])) += t;
```

Next

- Roofline model
- Mass flux divergence:

$$\mathrm{div}(v\Delta p)_{i,k} = \frac{1}{A_i} \sum_{l \in \mathcal{E}(i)} v_{n_l,k} \Delta \breve{p}_{l,k} (N_l \cdot n_{i,l}) l$$

  - fuse of operators: cell-to-edge averaging and div
  - k level: $\Delta p_k = \frac{1}{2}(p_{k-1/2} + p_{k+1/2})$