Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Home Affairs FDHA
Federal Office of Meteorology and Climatology MeteoSwiss

dusk & dawn - Advanced Concepts

Advanced Concepts on Unstructured Meshes

Advanced Concepts

Overview:

- Neighbour chains
- Sparse dimensions
  - Filling Sparse Dimension
  - Type Checking Sparse Dimensions
  - Horizontal Offsets
- Nesting of Reductions
- Index Fields
- Parallel model
  - Execution Safety
- Vertical Solvers

Neighbor Chains

- So far, the neighborhoods used in reductions were simple
- As a reminder, they can be enumerated by the following six cases

Edge →Cell	Cell →Edge	Node →Edge
Edge →Node	Cell →Node	Node →Cell

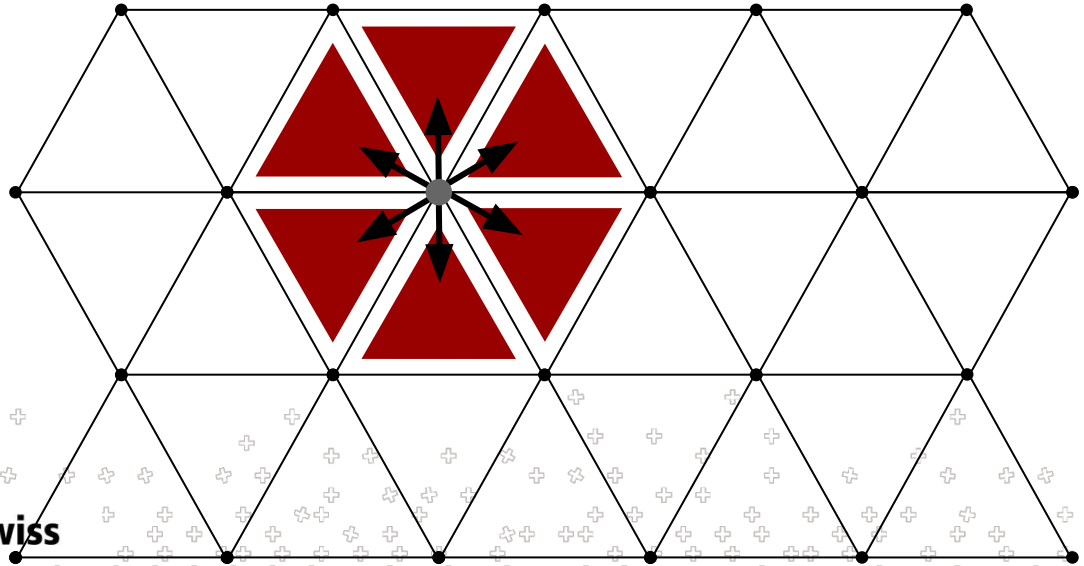
Neighbor Chains

- So far, the neighborhoods used in reductions were simple
- As a reminder, they can be enumerated by the following six cases

Edge →Cell
Edge →Node

Cell →Edge
Cell →Node

Node →Edge
Node →Cell



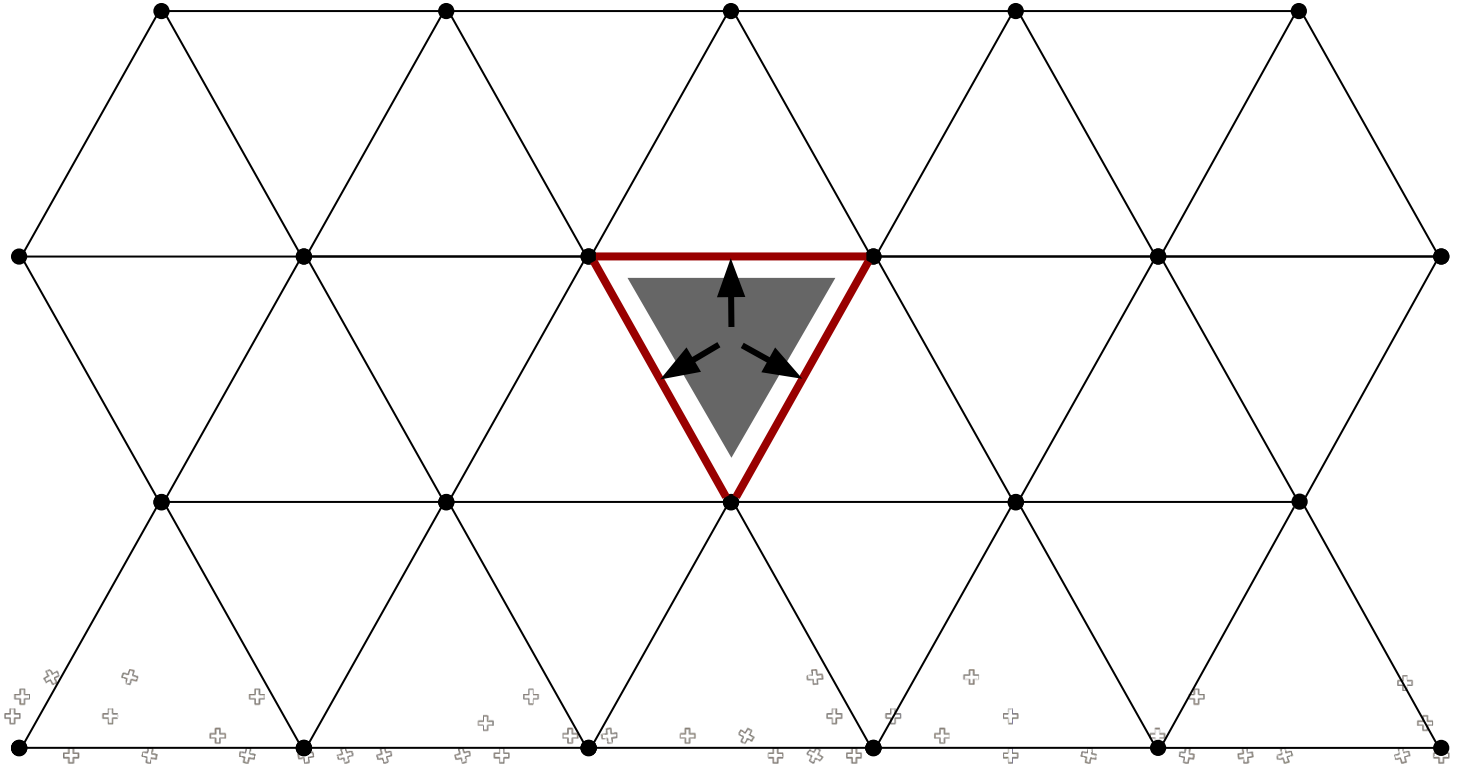
Neighbor Chains

- So far, the neighborhoods used in reductions were simple
- As a reminder, they can be enumerated by the following six cases

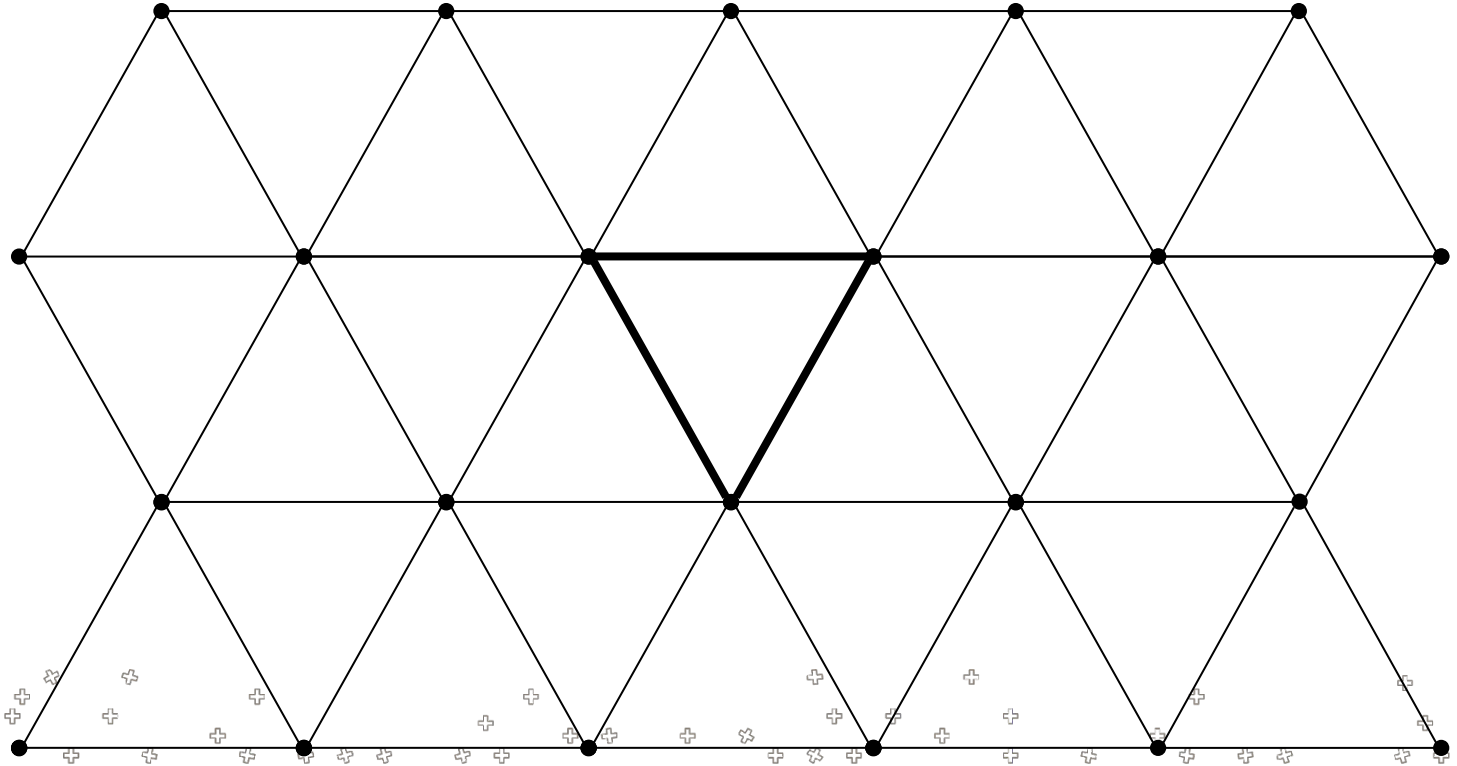
Edge →Cell	Cell →Edge	Node →Edge
Edge →Node	Cell →Node	**Node →Cell**

- It turns out that the ICON code (and surely a lot of other codes) uses more general neighborhoods
- Generalization of the neighborhood →**Neighbor Chains**

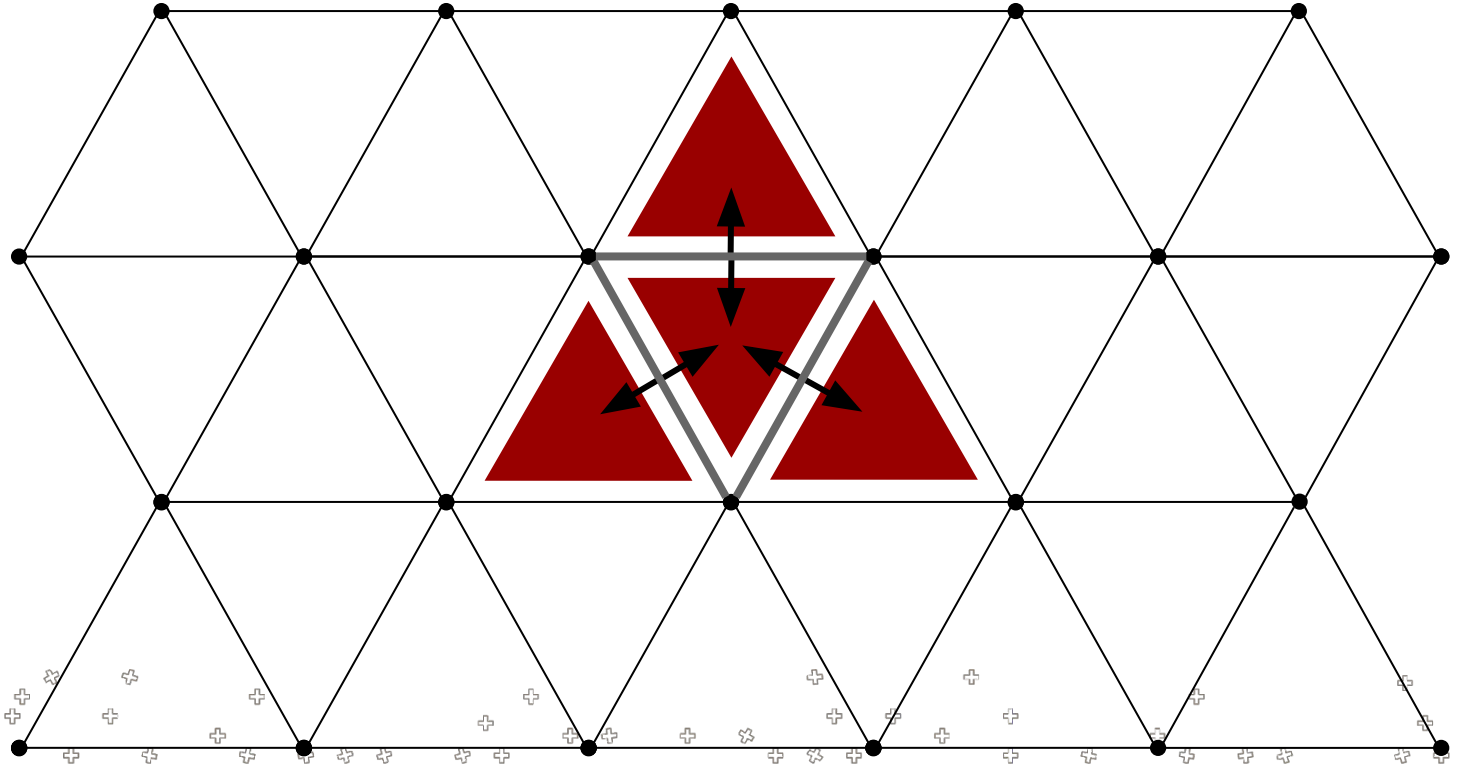
Neighbors: Cell -> Edge



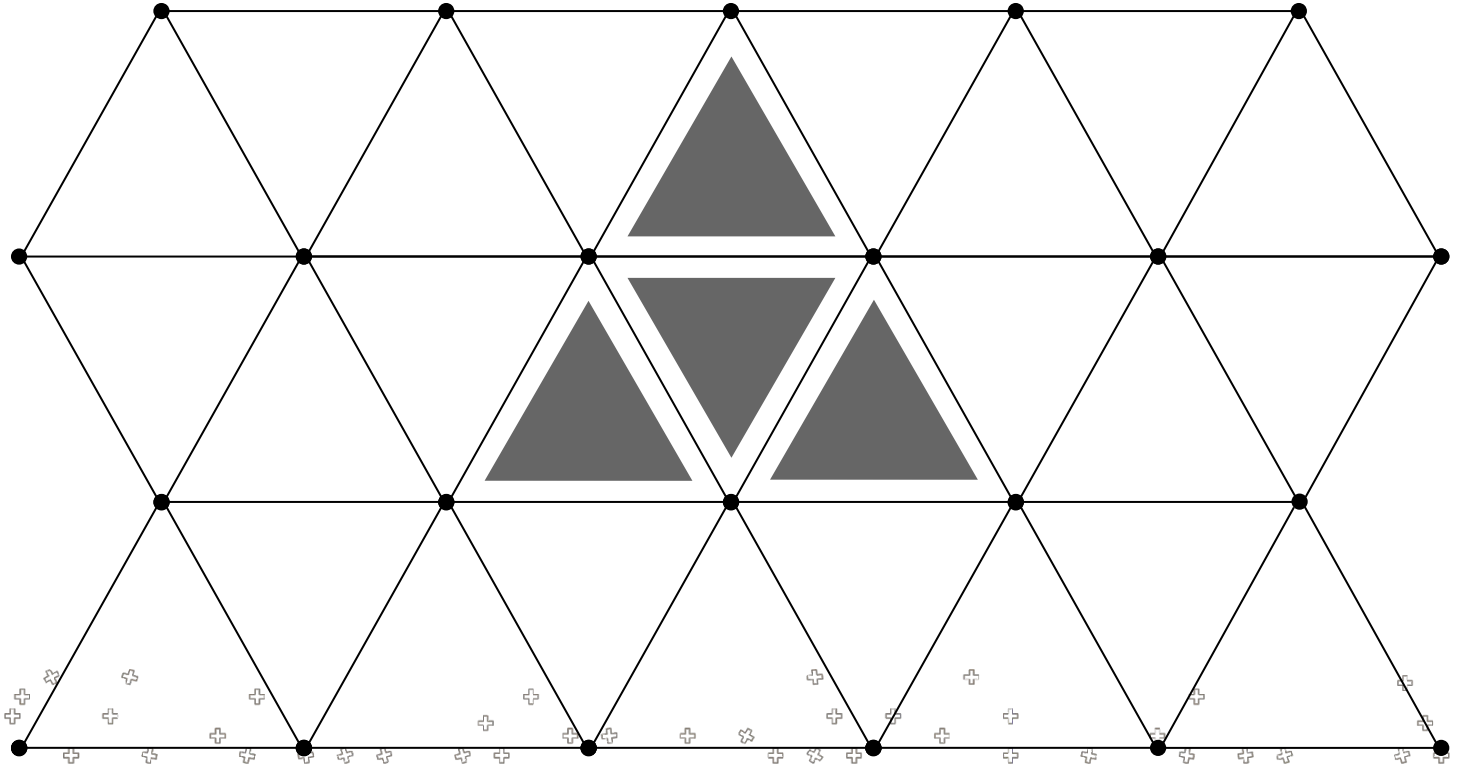
Neighbors: Cell -> Edge



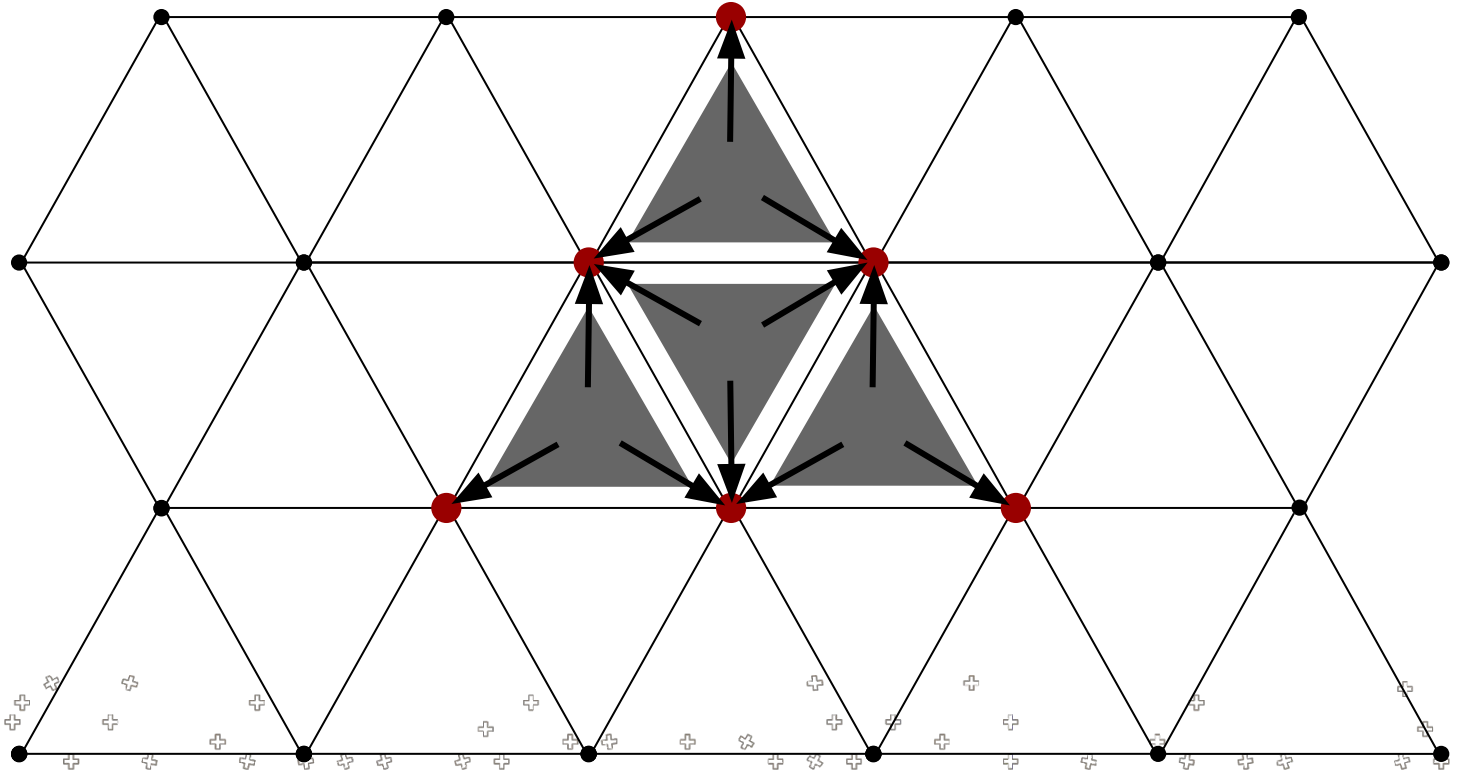
Neighbor Chain: Cell -> Edge -> Cell



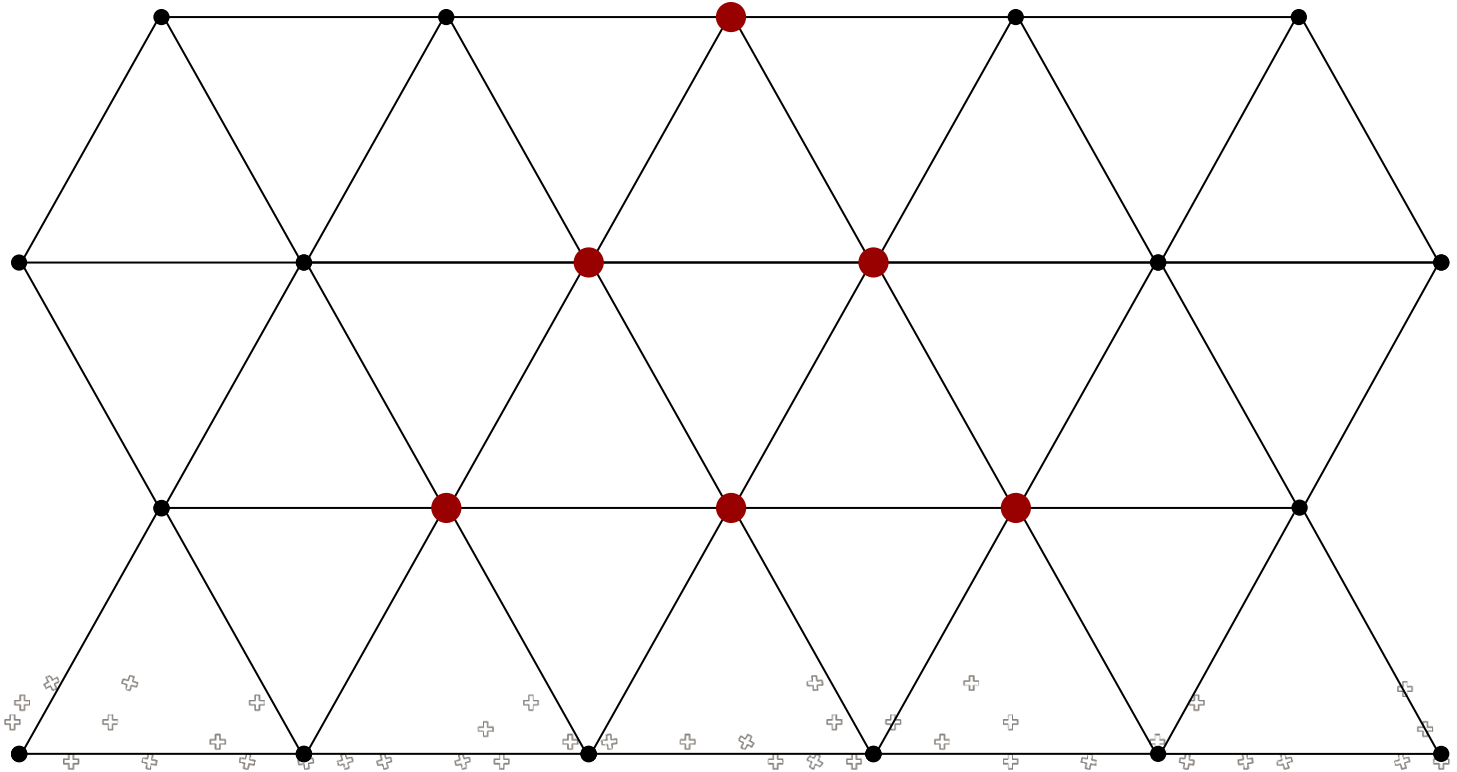
Neighbor Chain: Cell -> Edge -> Cell



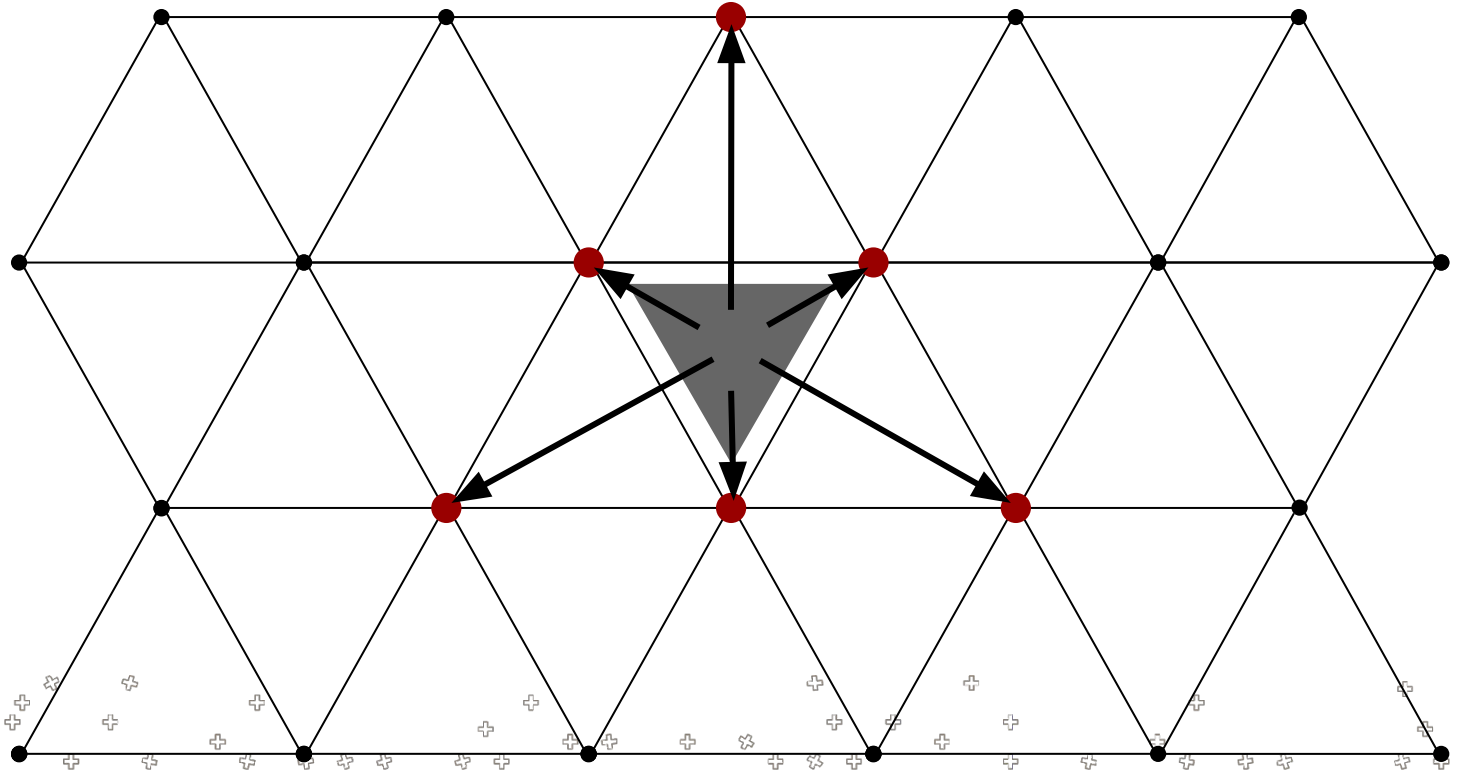
Neighbor Chain: Cell -> Edge -> Cell ->Vertex



Neighbor Chain: Cell -> Edge -> Cell ->Vertex



Neighbor Chain: Cell -> Edge -> Cell ->Vertex



Neighbor Chains - Use Cases

- In the ICON Dycore
  - High order interpolation
  - Finite Difference stencils on FVM meshes
- Other potential use cases
  - Wide FD stencils
  - ENO/WENO schemes
  - "Meshfree" Galerkin Methods
  - ...

Neighbor Chains - Case Study - RBF Interpolation

Specific use Case in the ICON dycore:

- Interpolate a quantity p_vn = $\mathbf{v} \cdot \mathbf{n}$ stored on edges to [p_u,p_v] located on cell centroids using **R**adial **B**asis **F**unctions
- Do not only consider edge values at direct edge neighbors
- Improve accuracy using a Neighbor Chain Instead

Neighbor Chains - Case Study - RBF Interpolation

FORTRAN Code

```
DO jk = slev, elev
 DO jc = i_startidx, i_endidx
   p_u_out(jc,jk,jb) =  &
     ptr_coeff(1,1,jc,jb)*p_vn_in(iidx(1,jc,jb),jk,iblk(1,jc,jb)) +
&
     ptr_coeff(2,1,jc,jb)*p_vn_in(iidx(2,jc,jb),jk,iblk(2,jc,jb)) +
&
     ptr_coeff(3,1,jc,jb)*p_vn_in(iidx(3,jc,jb),jk,iblk(3,jc,jb)) +
&
     ptr_coeff(4,1,jc,jb)*p_vn_in(iidx(4,jc,jb),jk,iblk(4,jc,jb)) +
&
     ptr_coeff(5,1,jc,jb)*p_vn_in(iidx(5,jc,jb),jk,iblk(5,jc,jb)) +
&
     ptr_coeff(6,1,jc,jb)*p_vn_in(iidx(6,jc,jb),jk,iblk(6,jc,jb)) +
&
     ptr_coeff(7,1,jc,jb)*p_vn_in(iidx(7,jc,jb),jk,iblk(7,jc,jb)) +
&
     ptr_coeff(8,1,jc,jb)*p_vn_in(iidx(8,jc,jb),jk,iblk(8,jc,jb)) +
&
```

Neighbor Chains - Case Study - RBF Interpolation

FORTRAN Code

```
DO jk = slev, elev
 DO jc = i_startidx, i_endidx
   p_u_out(jc,jk,jb) =  &
     ptr_coeff(1,1,jc,jb) *p_vn_in(iidx(1,jc,jb),jk,iblk(1,jc,jb)) +
&
     ptr_coeff(2,1,jc,jb) *p_vn_in(iidx(2,jc,jb),jk,iblk(2,jc,jb)) +
&
     ptr_coeff(3,1,jc,jb) *p_vn_in(iidx(3,jc,jb),jk,iblk(3,jc,jb)) +
&
     ptr_coeff(4,1,jc,jb) *p_vn_in(iidx(4,jc,jb),jk,iblk(4,jc,jb)) +
&
     ptr_coeff(5,1,jc,jb) *p_vn_in(iidx(5,jc,jb),jk,iblk(5,jc,jb)) +
&
     ptr_coeff(6,1,jc,jb) *p_vn_in(iidx(6,jc,jb),jk,iblk(6,jc,jb)) +
&
     ptr_coeff(7,1,jc,jb) *p_vn_in(iidx(7,jc,jb),jk,iblk(7,jc,jb)) +
&
     ptr_coeff(8,1,jc,jb) *p_vn_in(iidx(8,jc,jb),jk,iblk(8,jc,jb)) +
&
```

Neighbor Chains - Case Study - RBF Interpolation

equivalent dusk stencil

```
@stencil
def rbf_vec_interpol_cell(p_u_out: Field[Cell], p_v_out: Field[Cell], p_vn_in: Field[Edge]
                          ptr_coeff_x: Field[ Cell > Edge > Cell > Edge ], ptr_coeff_y: Field[ Cell > Edge > Cell >
Edge]):
    with levels_downward:
        p_u_out = sum_over(Cell > Edge > Cell > Edge, p_vn_in*ptr_coeff_x)
        p_v_out = sum_over(Cell > Edge > Cell > Edge, p_un_in*ptr_coeff_y)
```

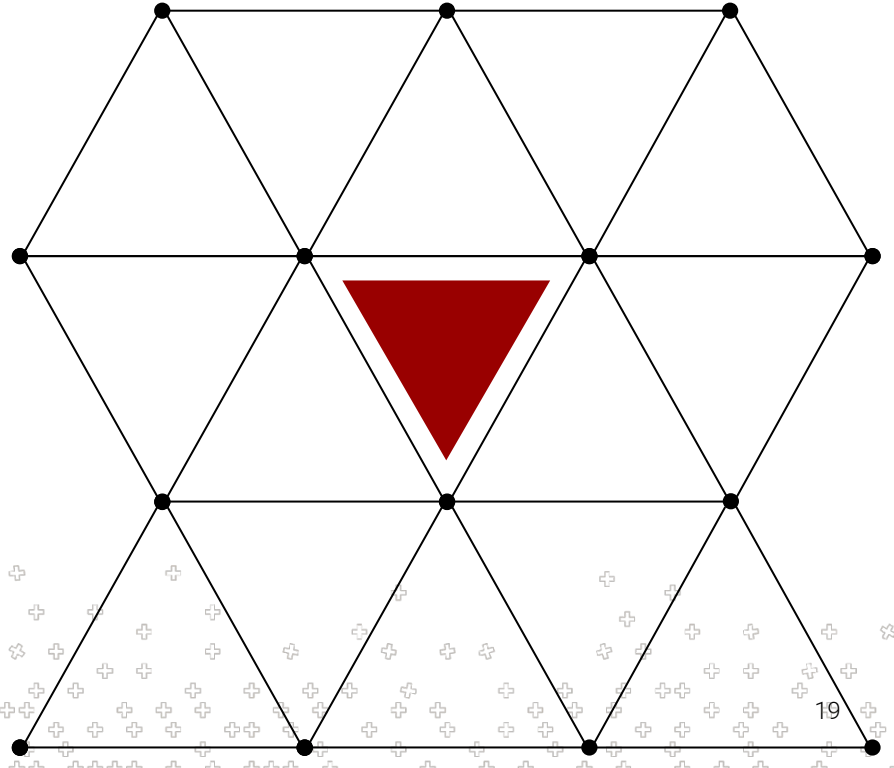
Neighbor Chains - Case Study - RBF Interpolation

- equivalent dusk stencil - shorten notation & remove v component for illustration

```
@stencil
def intp(u: Field[Cell], vn: Field[Edge], coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge, vn*coeff_x)
```

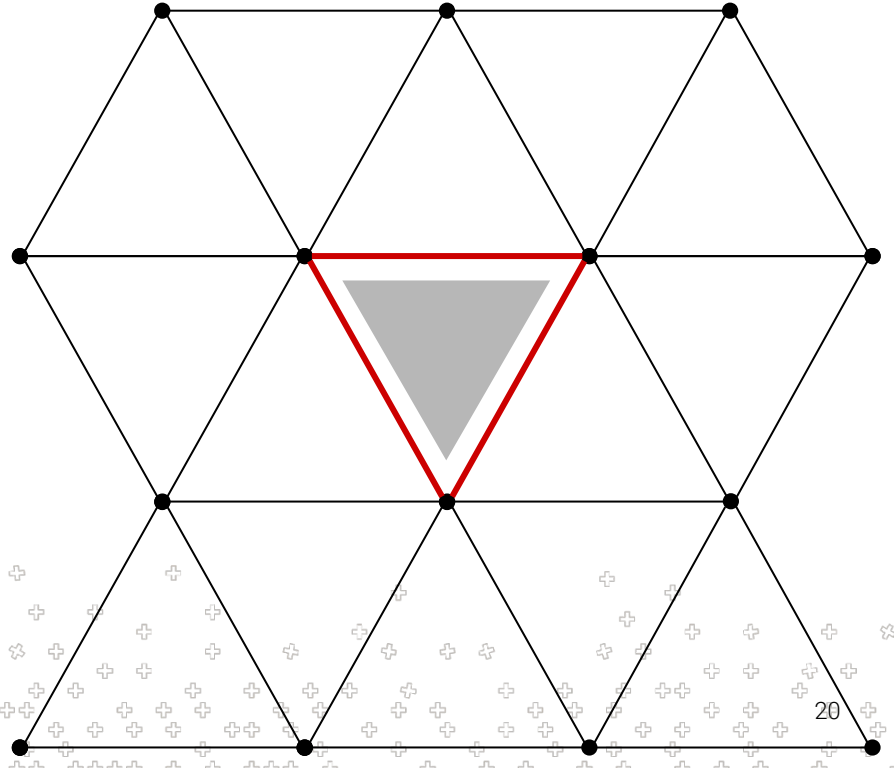
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



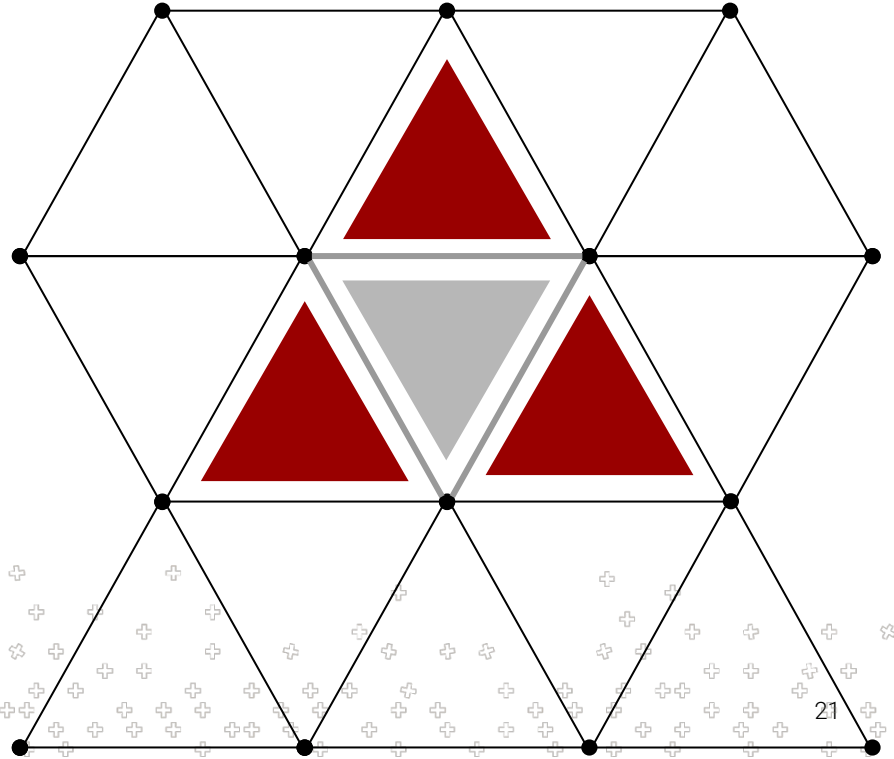
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



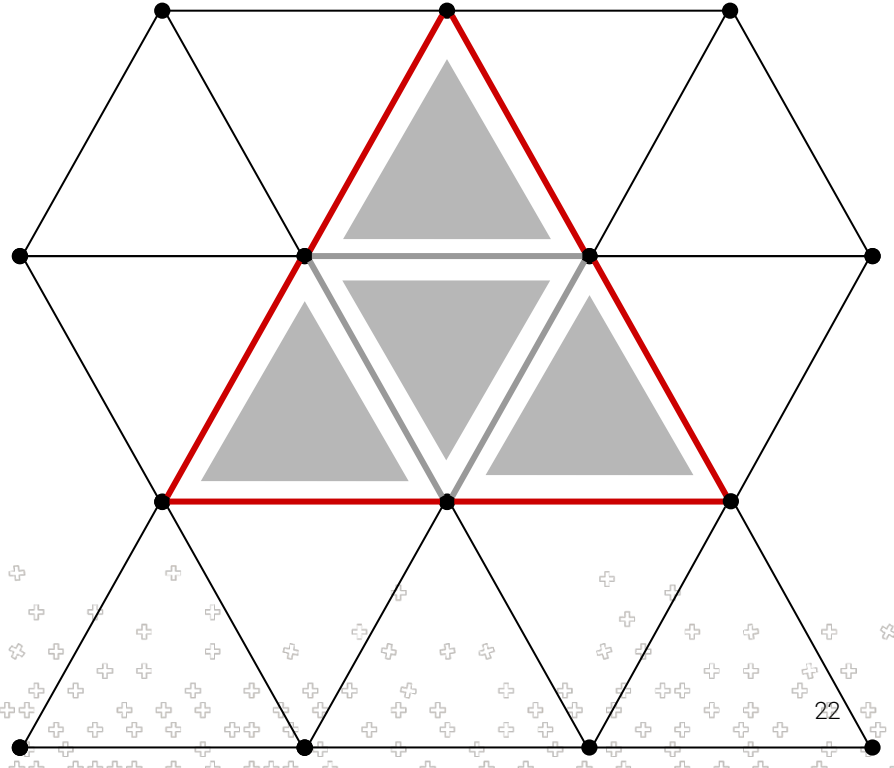
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



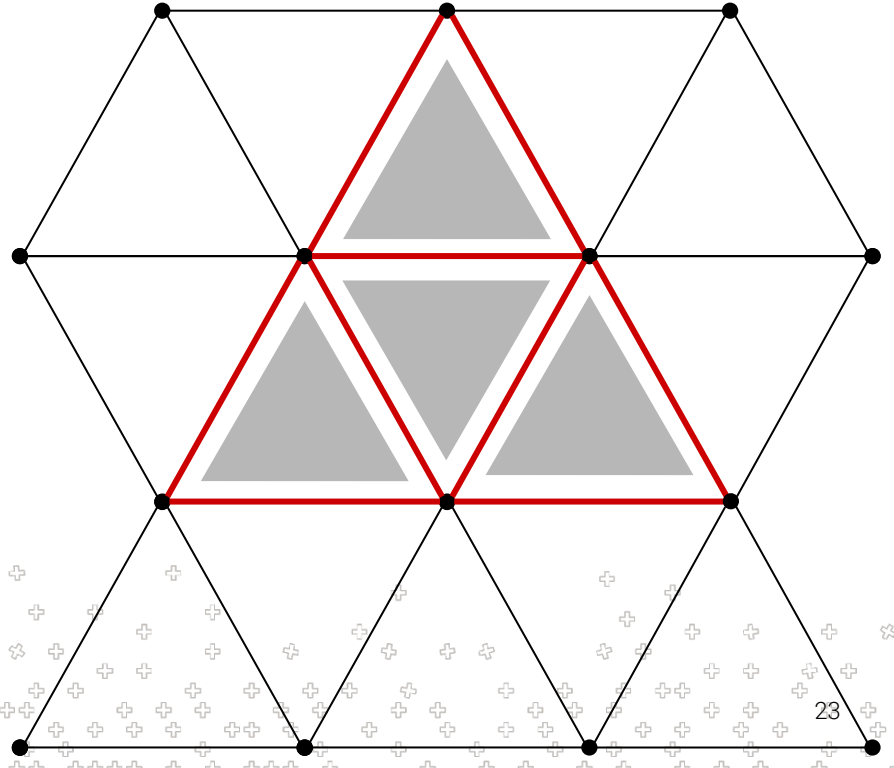
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



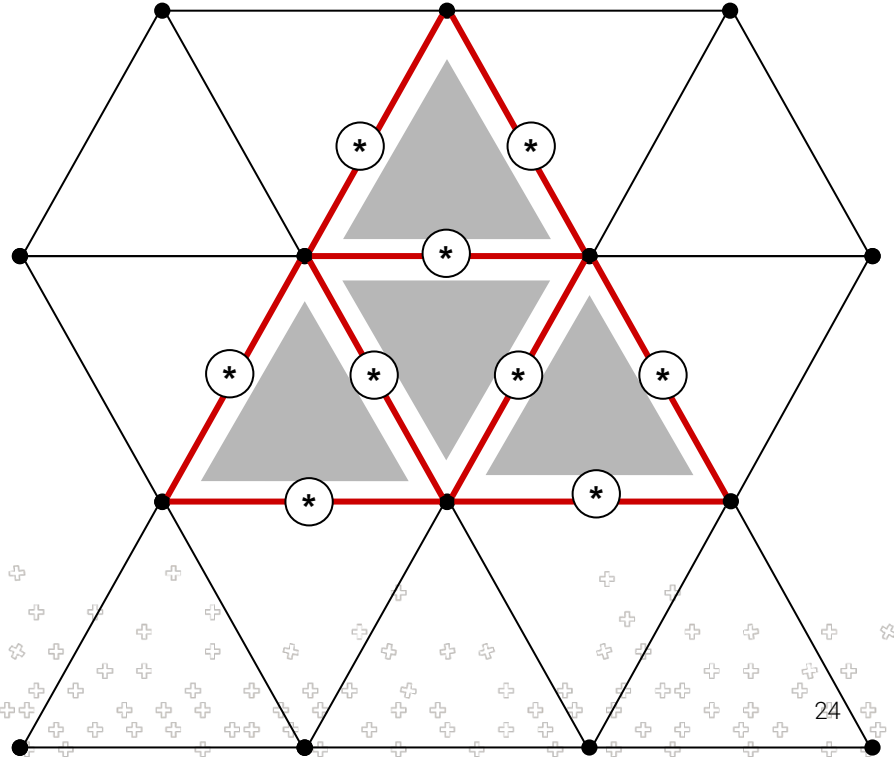
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



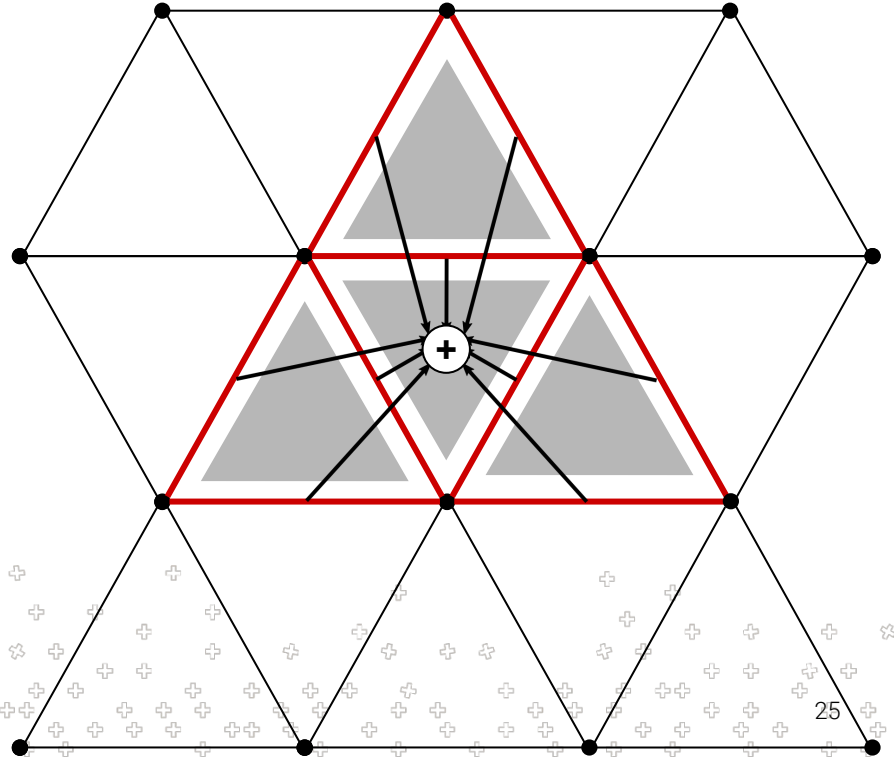
Neighbor Chains - Case Study - RBF Interpolation

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



Neighbor Chains - Case Study - RBF Interpolation

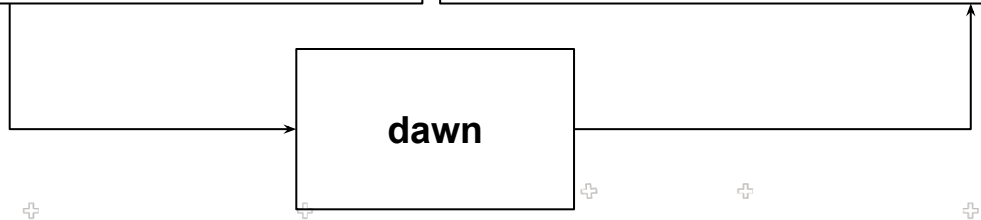
```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



Neighbor Chains - Emitted Pseudocode

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[...]):
   with levels_downward:
       u = sum_over(Cell > Edge > Cell > Edge,
                    vn*coeff_x)
```

```
parfor (cIdx = 0; cIdx < mesh.num_cells(); eIdx++)
    for (eIdx : mesh.nbh(cIdx, {Cell, Edge, Cell, Edge}))
        u(cIdx) += vn(eidx) * coeff_x(...)
```



Conventional Reduction - Emitted Pseudocode

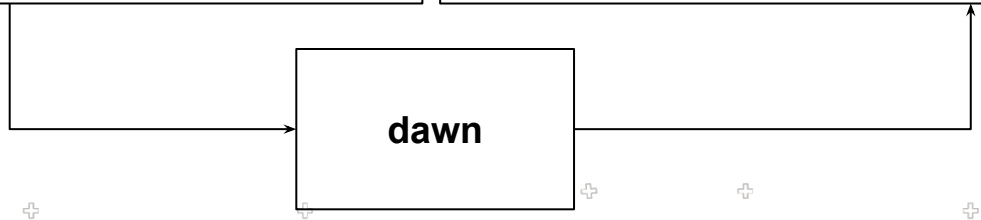
```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge]):
   with levels_downward:
       u = sum_over(Cell > Edge, vn)
```

```
parfor (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
    for (eIdx : mesh.nbh(cIdx, {Cell, Edge})
          u(cIdx) += vn(eIdx)
```

Conventional Reduction - Emitted Pseudocode

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge]):
   with levels_downward:
       u = sum_over(Cell > Edge, vn)
```

```
parfor (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
    for (eIdx : mesh.nbh_edges(cIdx))
        u(cIdx) += vn(eIdx)
```

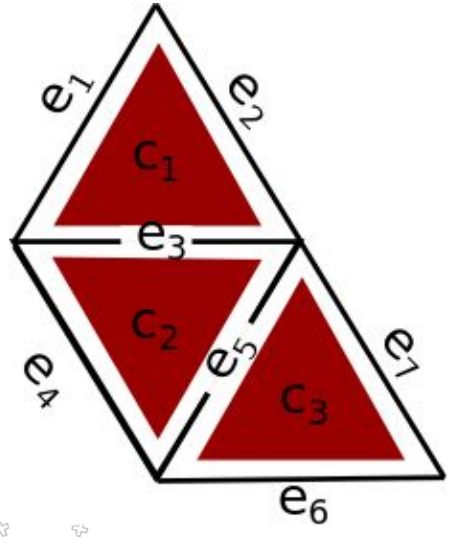


Sparse Dimensions

- Note "Sparse Dimension" is not standard terminology (made up by MCH)
- Essentially: arrays of higher rank which store values for each neighbor of a cell/edge/node
  - extends to neighbor chains
- Called "Sparse Dimensions" because they **can be** implemented as sparse matrices
  - like an adjacency matrix but stores a value instead of 0/1 for not connected / connected

Sparse Dimensions

Mesh



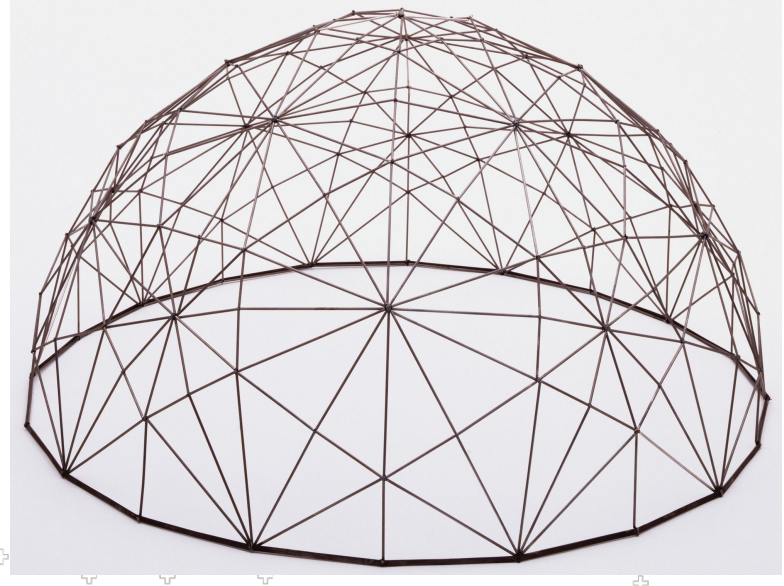
Adjacency Matrix

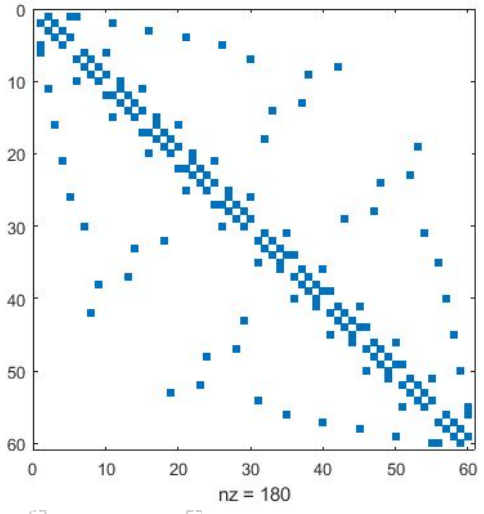
Edge / **Cell**

1	0	0
1	0	0
1	1	0
0	1	0
0	1	1
0	0	1
0	0	1

Sparse Dimensions

"Mesh"





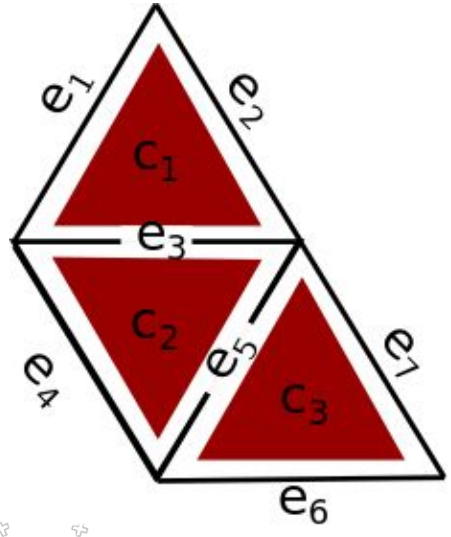
Adjacency Matrix

*R. Buckminster Fuller Geodesic Dome 1952

~5% of entries non zero

Sparse Dimensions

Mesh

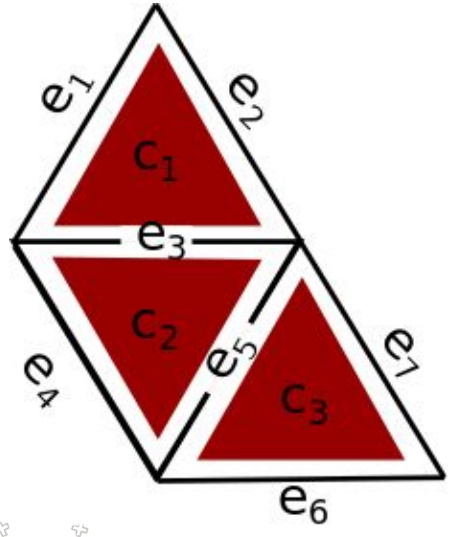


Sparse Dimension: dist_to_c (Matrix Representation)

Edge \ Cell			
	√3	∅	∅
	√3	∅	∅
	√3	√3	∅
	∅	√3	∅
	∅	√3	√3
	∅	∅	√3
	∅	∅	√3

Sparse Dimensions

Mesh

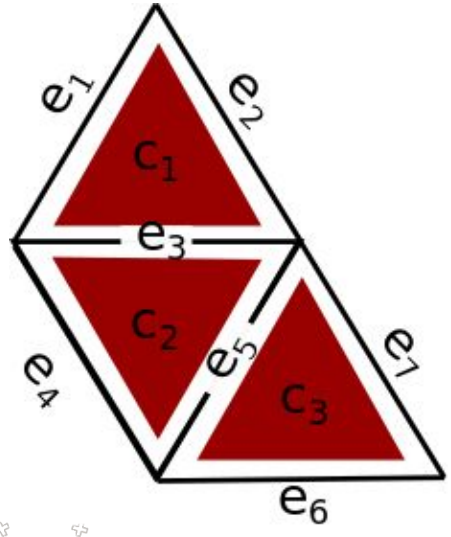


Sparse Dimension: dist_to_c (Array Representation)

```
dist_to_c(e1,:) = {√3,   ∅}
dist_to_c(e2,:) = {√3,   ∅}
dist_to_c(e3,:) = {√3,  √3}
dist_to_c(e4,:) = {√3,   ∅}
dist_to_c(e5,:) = {√3,  √3}
dist_to_c(e6,:) = {√3,   ∅}
dist_to_c(e7,:) = {√3,   ∅}
```

Sparse Dimensions

Mesh



two cells per edge → two entries

Sparse Dimension: dist_to_c (Array Representation)

```
dist_to_c(e1,:) = {√3,   ∅}
dist_to_c(e2,:) = {√3,   ∅}
dist_to_c(e3,:) = {√3,  √3}
dist_to_c(e4,:) = {√3,   ∅}
dist_to_c(e5,:) = {√3,  √3}
dist_to_c(e6,:) = {√3,   ∅}
dist_to_c(e7,:) = {√3,   ∅}
```

Sparse Dimensions

three cells in the mesh → three entries

Adjacency Matrix

Edge \ Cell			
	1	0	0
	1	0	0
	1	1	0
	0	1	0
	0	1	1
	0	0	1
	0	0	1

two cells per edge → two entries

Sparse Dimension: dist_to_c (Array Representation)

$$
\begin{aligned}
\texttt{dist\_to\_c}(e_1,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_2,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_3,:) &= \{\sqrt{3}, \sqrt{3}\} \\
\texttt{dist\_to\_c}(e_4,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_5,:) &= \{\sqrt{3}, \sqrt{3}\} \\
\texttt{dist\_to\_c}(e_6,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_7,:) &= \{\sqrt{3}, \varnothing\}
\end{aligned}
$$

Sparse Dimensions

three cells in the mesh → three entries

Adjacency Matrix

Edge \ Cell			
	1	0	0
	1	0	0
	1	1	0
	0	1	0
	0	1	1
	0	0	1
	0	0	1

two cells per edge → two entries

Sparse Dimension: dist_to_c (Array Representation)

$$
\begin{aligned}
\texttt{dist\_to\_c}(e_1,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_2,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_3,:) &= \{\sqrt{3}, \sqrt{3}\} \\
\texttt{dist\_to\_c}(e_4,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_5,:) &= \{\sqrt{3}, \sqrt{3}\} \\
\texttt{dist\_to\_c}(e_6,:) &= \{\sqrt{3}, \varnothing\} \\
\texttt{dist\_to\_c}(e_7,:) &= \{\sqrt{3}, \varnothing\}
\end{aligned}
$$

- Array representation does not include connectivity
- But is (way) more compact

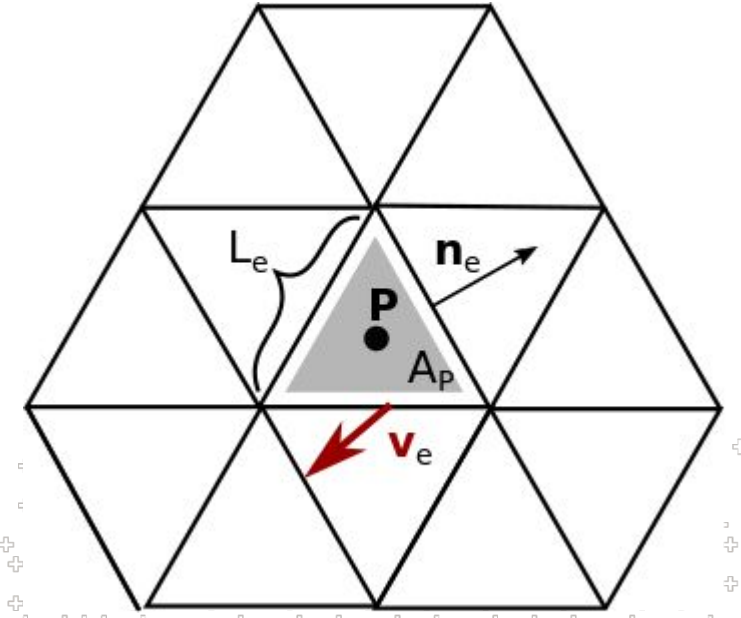
Sparse Dimensions

- Called "Sparse Dimensions" because they **can be** implemented as sparse matrices
  - like an adjacency matrix but stores a value / a (NULL) type instead of 0/1 for not connected / connected
- More reasonable implementation is array form discussed
  - no need to replicate connectivity information into each sparse dimension

Sparse Dimensions - Example Use Case #1 - Geometrical Factors

- You have already used a sparse dimension in the differential operators exercise, albeit in "black box" fashion
- Now it's time to understand them properly
- Remember the equation for the divergence

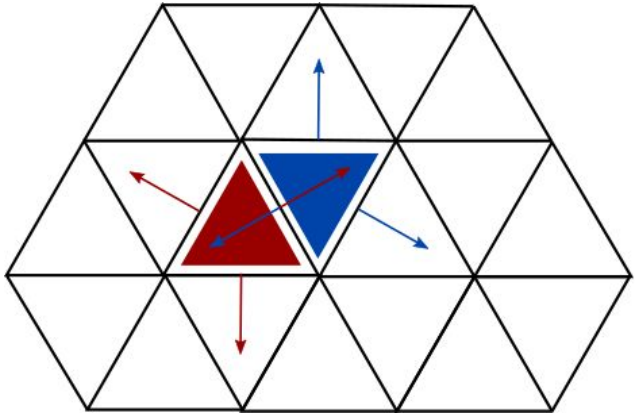
$$\langle \nabla \cdot \mathbf{v}(\mathbf{P}) \rangle_{FVM} = \frac{1}{A_P} \sum_{e=1}^{3} (\mathbf{v}_e \cdot \mathbf{n}_e) L_e$$



- Note that this equation only holds if the normals $\mathbf{n}_f$ point outside the triangle with centroid P

Sparse Dimensions - Example Use Case #1 - Geometrical Factors

- The equation for the gradient on the last slide holds only if all normals point outside
- However, one clearly can't arrange a mesh such that all normals point outside

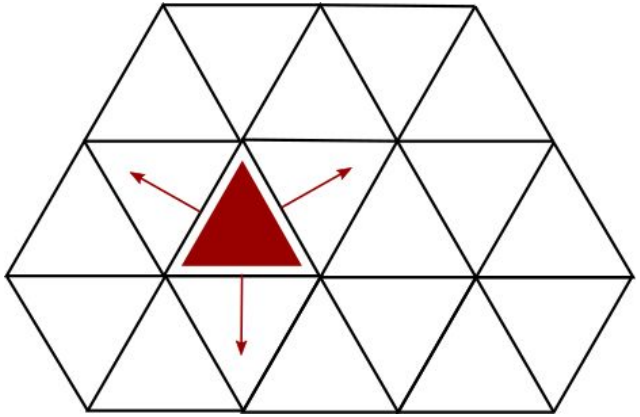


- To check, for each normal, in which side it points during the computation of the gradient is not efficient (involves the evaluation of a dot product)
- Usually, this is solved by pre-computing a geometrical factor. In this case three signs for each triangle, that indicate whether the normal is flipped during the summation
- This is a prime example of a sparse field

Sparse Dimensions - Example Use Case #1 - Geometrical Factors

- The equation for the gradient on the last slide holds only if all normals point outside
- However, one clearly can't arrange a mesh such that all normals point outside

edge_orientation(**cell_a**,:) = [1, 1, 1]

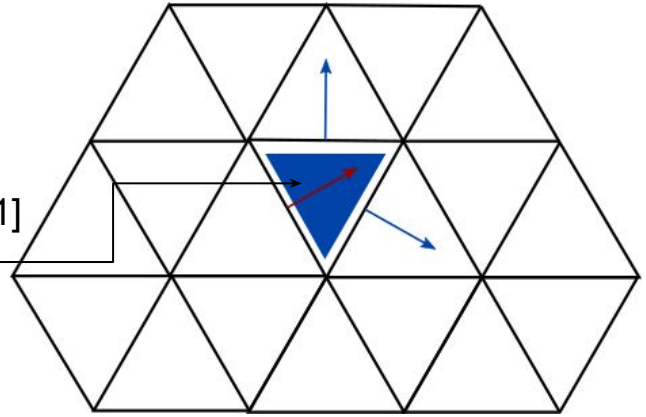


- To check, for each normal, in which side it points during the computation of the gradient is not efficient (involves the evaluation of a dot product)
- Usually, this is solved by pre-computing a geometrical factor. In this case three signs for each triangle, that indicate whether the normal is flipped during the summation
- This is a prime example of a sparse dimension

Sparse Dimensions - Example Use Case #1 - Geometrical Factors

- The equation for the gradient on the last slide holds only if all normals point outside
- However, one clearly can't arrange a mesh such that all normals point outside

edge_orientation(**cell_b**,:) = [**-1**, 1, 1]



- To check, for each normal, in which side it points during the computation of the gradient is not efficient (involves the evaluation of a dot product)
- Usually, this is solved by pre-computing a geometrical factor. In this case three signs for each triangle, that indicate whether the normal is flipped during the summation
- This is a prime example of a sparse dimension

Sparse Dimensions - Example Use Case #2 - Interpolation Coefficients

- There are myriads of interpolation schemes
- Roughly speaking they:
  - all consider a few close neighbor points
  - weight these points with weights w_i which need to sum up to 1 $\sum_{i=1}^{N} w_i = 1$
  - the weights are related to the distance of the neighboring points i
- Using a suitable kernel function W a broad case of interpolation problems can be mapped to

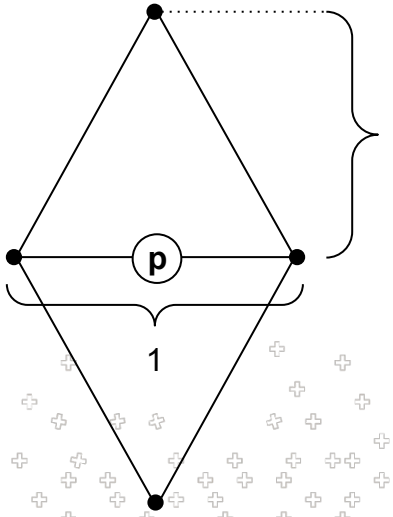
$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

Sparse Dimensions - Example Use Case #2 - Interpolation Coefficients

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

- In the case of a perfectly regular mesh with periodic boundaries (i.e. the $\mathbf{q}_i$ are equal for all $\mathbf{p}$) there are only |N(**p**)| different weights, and they can, in fact, be expressed using the weights concept. E.g.

```
@stencil
def equidist_interpol(f_intp: Field[Edge], f: Field[Vertex]):
    with levels_downward:
        f_intp = sum_over(Edge > Cell > Vertex, f,
                          weights = [2, 2, 2/√3, 2/√3]) / (4 + 4√3)
```



Sparse Dimensions - Example Use Case #2 - Interpolation Coefficients

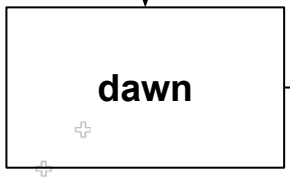
$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

- However, if this is not the case (as in most any practical application), then the weights are different for each point p, and a sparse dimension is required

```
@stencil
def equidist_interpol(f_intp: Field[Edge], f: Field[Vertex], coeffs: Field[Edge > Cell > Vertex]):
    with levels_downward:
        f_intp = sum_over(Edge > Cell > Vertex, f*coeffs)
```

Sparse Dimensions - Emitted Pseudocode

```
@stencil
def equidist_interpol(f_intp: Field[Edge,K],
                      fv: Field[Vertex,K],
                      coeffs: Field[Edge>Cell>Vertex]):
   with levels_downward:
       f_intp = sum_over(Edge > Cell > Vertex, f*coeffs)
```

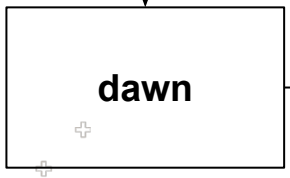



```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          f_intp(eIdx,k) += fv(vidx,k)*coeffs(eIdx, linear_idx)
          linear_idx++
      }
  }
}
```

MeteoSwiss

Sparse Dimensions - Emitted Pseudocode

```
@stencil
def equidist_interpol(f_intp: Field[Edge, K],
                      fv: Field[Vertex, K],
                      coeffs: Field[Edge>Cell>Vertex,K]):
   with levels_downward:
       f_intp = sum_over(Edge > Cell > Vertex, f*coeffs)
```

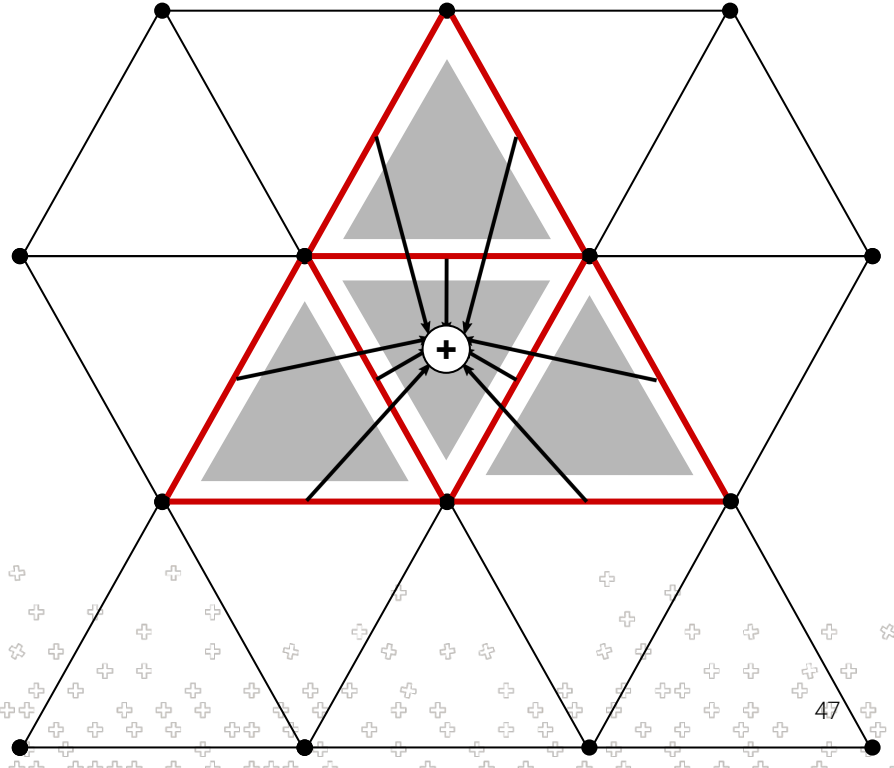



```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        f_intp(eIdx,k) += fv(vidx,k)*coeffs(eIdx, k, linear_idx)
        linear_idx++
      }
  }
}
```

MeteoSwiss

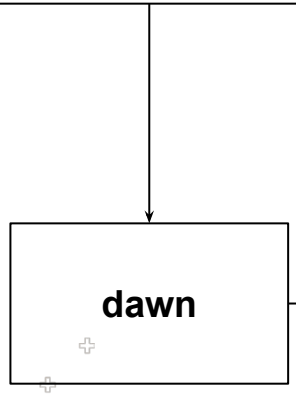
Sparse Dimensions - Emitted Pseudocode

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
    with levels_downward:
        u = sum_over(Cell > Edge > Cell > Edge,
                     vn*coeff_x)
```



Sparse Dimensions - Emitted Pseudocode

```
@stencil
def intp(u: Field[Cell],
         vn: Field[Edge],
         coeff_x: Field[Cell > Edge > Cell > Edge]):
   with levels_downward:
       u = sum_over(Cell > Edge > Cell > Edge,
                    vn*coeff_x)
```




```
parfor (k = 0; k < kmax; k++) {
  parfor (cIdx = 0; cIdx < mesh.num_cells(); cIdx++) {
      linear_idx = 0
      for (eIdx : mesh.nbh(cIdx,{Cell > Edge > Cell > Edge})) {
          u(cIdx,k) += vn(eidx,k)*coeff_x(cIdx, linear_idx)
          linear_idx++
      }
  }
}
```

Q&A

Filling Sparse Dimensions

- So far, sparse dimensions have just been assumed as being pre-filled with the appropriate values
- There is nothing "magic" about this, if dusk & dawn wouldn't offer a means to fill sparse dimensions this would simply mean that it's the responsibility of the driver code to do so
- This choice may be appropriate if the sparse dimensions consist of quantities dependent on the mesh geometry only. So it can be precomputed once in an initialization step and then used throughout the simulation
    - This is due to the fact that the mesh doesn't deform in Eulerian simulations (e.g. climate applications)

- In ICON this is **NOT** the case, i.e. there are sparse fields dependent on the current velocities for example

→ dusk offers a concept to do so: `with sparse[CHAIN]:`

Filling Sparse Dimensions - Real World Example

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

Filling Sparse Dimensions - Real World Example

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

sparse index

Filling Sparse Dimensions - Real World Example

Neighbor Table for **Edge > Cell > Vertex**

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

Filling Sparse Dimensions - Real World Example - Dusk Version

```
@stencil
def ICON_laplacian_diamond(
    u_vert: Field[Vertex, K],
    v_vert: Field[Vertex, K],
    primal_normal_x: Field[Edge > Cell > Vertex],
    primal_normal_y: Field[Edge > Cell > Vertex],
    vn_vert: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn_vert = u_vert * primal_normal_x + v_vert * primal_normal_y
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (nIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
            vn(vIdx, linear_idx, k) +=
                u(vIdx,k)*nx(eIdx, linear_idx)
                v(vIdx,k)*ny(eIdx, linear_idx)
            linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (nIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx)
              v(vIdx,k)*ny(eIdx, linear_idx)
          linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx)
              v(vIdx,k)*ny(eIdx, linear_idx)
          linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx, k)
              v(vIdx,k)*ny(eIdx, linear_idx, k)
          linear_idx++
      }
  }
}
```

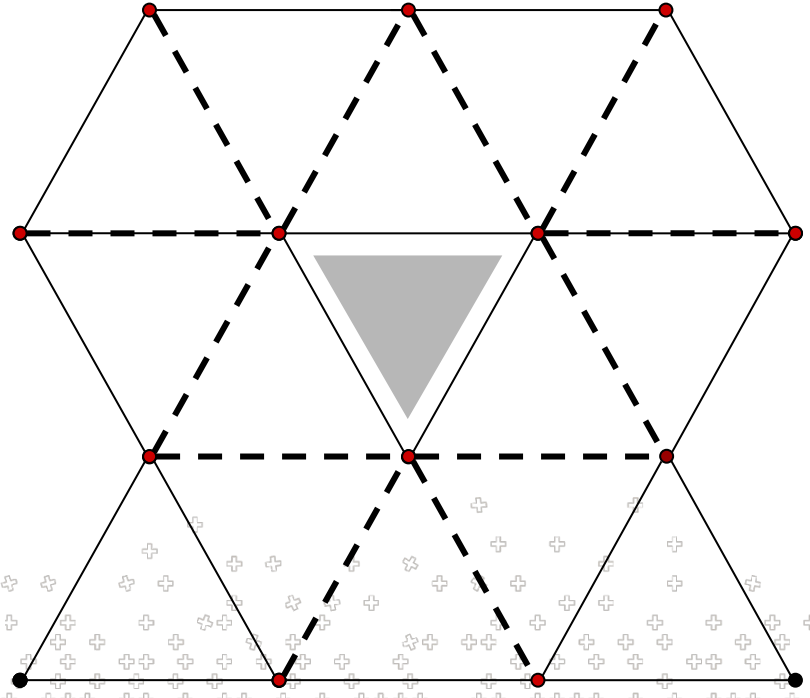
Filling Sparse Dimensions - Computing Interpolation Coefficients

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

- Coeffs depend on positions only, so this could be handled in driver code
- Still an interesting case to look at
- Works for arbitrary neighborhoods. Let's pick one we didn't consider so far: Cell -> Vertex -> Edge -> Vertex

Filling Sparse Dimensions - Computing Interpolation Coefficients

Cell -> Vertex -> **Edge** -> Vertex



Filling Sparse Dimensions - Computing Interpolation Coefficients

$$\langle f(\mathbf{p})\rangle_{\text{intp}} = \frac{1}{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)} \boxed{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)\cdot f(\mathbf{q}_i)}$$

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

$$W_h(\mathbf{p}-\mathbf{q}_i) = \frac{1}{h\pi^2} e^{\left(\frac{||\mathbf{p}-\mathbf{q}_i||^2}{h^2}\right)}$$

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

`with` statement encloses a *block* of statements
→can introduce intermediary results

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

such temporary fields need to be declared (for now)

`with` statement encloses a *block* of statements
→can introduce intermediary results

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

no support for globals / scalar arguments yet, need to waste a lot of memory

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p})\rangle_{\text{intp}} = \frac{1}{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)} \sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)\cdot f(\mathbf{q}_i)$$

```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex],
        wij: Field[Cell > Vertex > Edge > Vertex], Wn: Field[Cell]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex]):
    wij: Field[Cell > Vertex > Edge > Vertex]
    qij: Field[Cell > Vertex > Edge > Vertex]
    Wn: Field[Cell]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

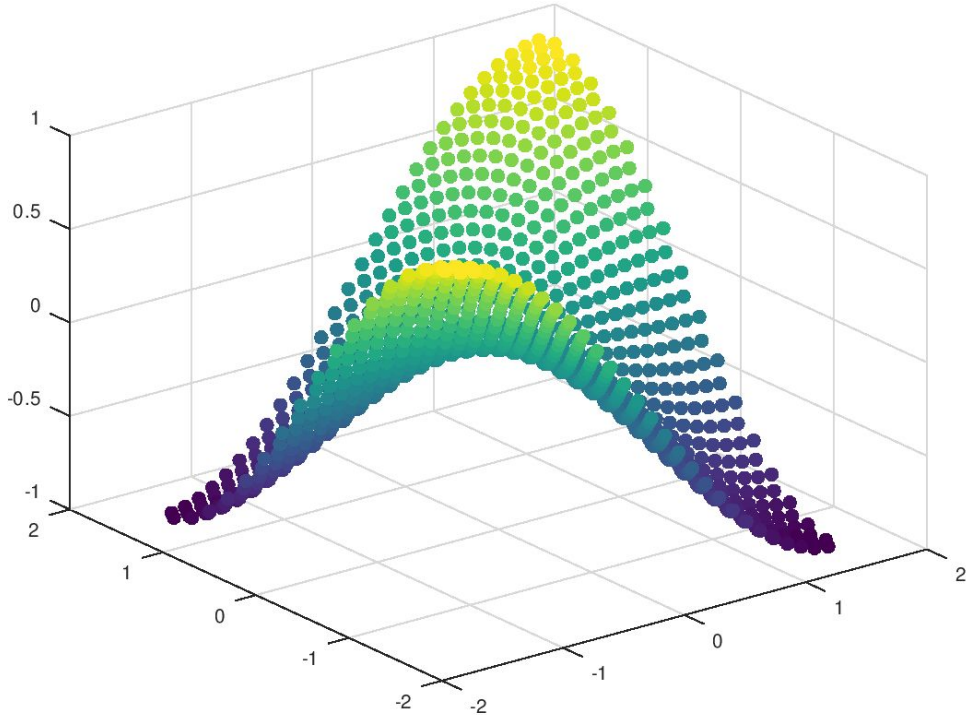
```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex]):
    wij: Field[Cell > Vertex > Edge > Vertex]
    qij: Field[Cell > Vertex > Edge > Vertex]
    Wn: Field[Cell]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

```
for (int kIter = klo; kIter < khi; kIter++) {
  if (kIter >= kSize + 0) { return; }
  ::dawn::float_type lhs_157 = (::dawn::float_type)0;
  for (int nbhIter = 0; nbhIter < C_V_E_V_SIZE; nbhIter++) {
    int nbhIdx = cvevTable[pidx * C_V_E_V_SIZE + nbhIter];
    if (nbhIdx == DEVICE_MISSING_VALUE) { continue;}
    lhs_157 += wij[nbhIter * NumCells + pidx];
  }
  ::dawn::float_type __local_Wn_1_115 = lhs_157;
  ::dawn::float_type lhs_161 = (::dawn::float_type)0;
  for (int nbhIter = 0; nbhIter < C_V_E_V_SIZE; nbhIter++) {
    int nbhIdx = cvevTable[pidx * C_V_E_V_SIZE + nbhIter];
    if (nbhIdx == DEVICE_MISSING_VALUE) { continue; }
    lhs_161 +=
        ((((int)1 / __local_Wn_1_115) * wij[nbhIter * NumCells + pidx]) *
         fn[nbhIdx]);
  }
  f_intp[pidx] = lhs_161;
```

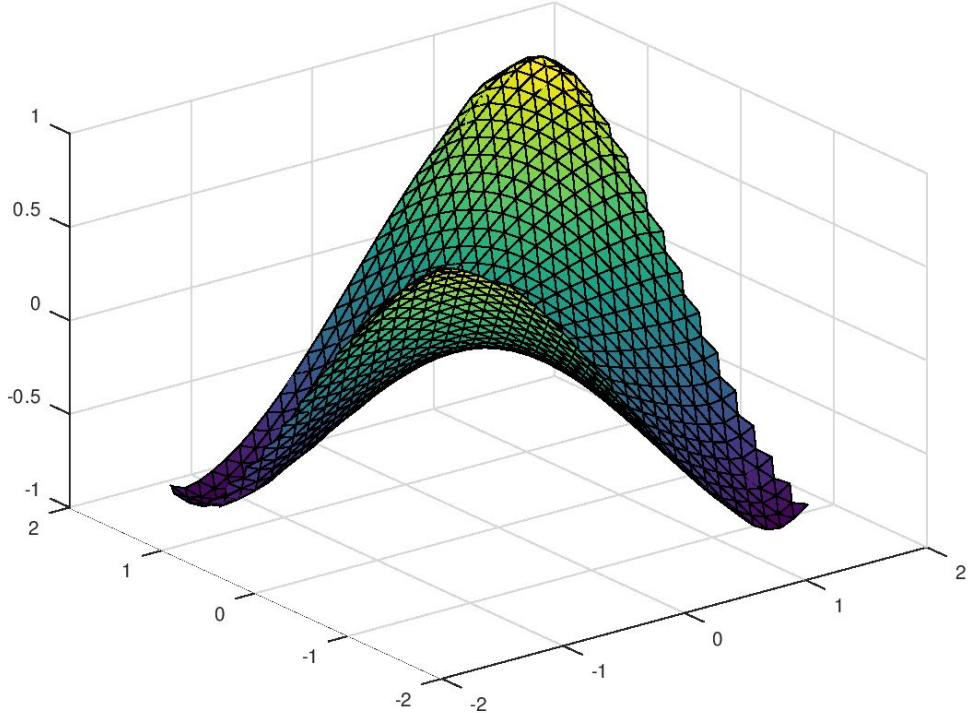
Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot \boxed{f(\mathbf{q}_i)}$$



Quick Excursion: Writing the complete interpolation

$$\boxed{\langle f(\mathbf{p}) \rangle}_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$



Sparse Dimensions and their Type

- We have seen quite a few examples of sparse dimensions now
- Let's take a step back and think about their (location) type
- An ordinary field has a single (location) type

```
eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex]
```

- A sparse field has a chain of locations as (location) type

```
eceField: Field[Edge > Cell > Edge], ecvField: Field[Edge > Cell > Vertex]
```

Sparse Dimensions and their Type

- A statement involving ordinary fields only (and is consistent in it's type) is generated into a loop over it's location type

```
@stencil
def dense(
    a: Field[Edge], b: Field[Edge],
    c: Field[Cell], d: Field[Cell]):
    with levels_downward:
        a = b
        c = d
```



```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
     a(eIdx) = b(eIdx)
for (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
     c(cIdx) = d(cIdx)
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = b*sparseF
```

?

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
  linear_idx = 0
  for (nIdx : mesh.nbh(eIdx, {Edge, Cell})) {
    a(eIdx) = b(eIdx)*sparseF(eIdx,linear_idx)
    linear_idx++
  }
}
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = b*sparseF
```

?

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
  linear_idx = 0
  for (nIdx : mesh.nbh(eIdx, {Edge, Cell})) {
    a(eIdx) += b(eIdx)*sparseF(eIdx,linear_idx)
    linear_idx++
  }
}
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A with sparse statement

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A **reduction** expressions
  - A `with sparse` statement

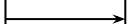
```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = sum_over(Edge>Cell, b*sparseF)
```

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
  linear_idx = 0
  for (cIdx : mesh.nbh(eIdx, {Edge, Cell})) {
     a(eIdx) += b(eIdx)*sparseF(eIdx,linear_idx)
     linear_idx++
  }
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement

```
@stencil
def sparse(
   a: Field[Edge], b: Field[Edge],
   sparseF: Field[Edge>Cell]):
   with levels_downward:
     with sparse[Edge>Cell]:
       sparseF = b*a
```



```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
  linear_idx = 0
  for (cIdx : mesh.nbh(eIdx, {Edge, Cell})) {
     sparseF(eIdx,linear_idx) = a(eIdx) * b(eIdx)
     linear_idx++
  }
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the sparse dimension stated in the first argument of the reduction / the parameter of the `with sparse` statement.

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the **sparse dimension** stated in the **first argument of the reduction** / the parameter of the `with sparse` statement.

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge], sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = sum_over(Edge>Cell, b*sparseF)
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the **sparse dimension** stated in the first argument of the reduction / the **parameter of the** `with sparse` statement.

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge], sparseF: Field[Edge>Cell]):
    with levels_downward:
      with sparse[Edge>Cell]:
        sparseF = b*a
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * ?)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * ?)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * eField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * vField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * cField)
```

ILLEGAL CODE

Sparse Dimensions and their Type

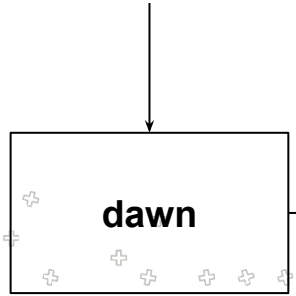
- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * eField *
                                                   vField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



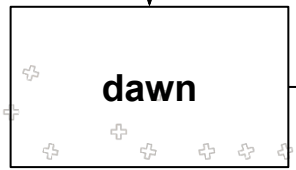
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx, k)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



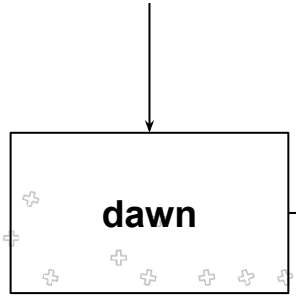
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



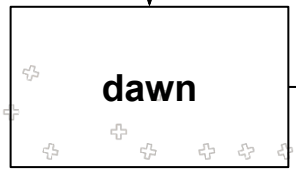
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], nField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
    with levels_downward:
        eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                      sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex]
):
    with levels_downward:
        with sparse[Edge > Cell > Vertex]:
            sparseF = eField*vField
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex]
):
    with levels_downward:
        with sparse[Edge > Cell > Vertex]:
            sparseF = cField
```

ILLEGAL CODE

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```

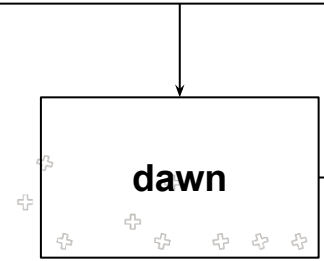

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



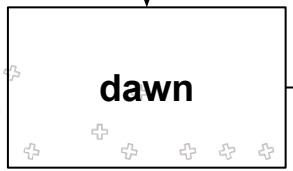

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



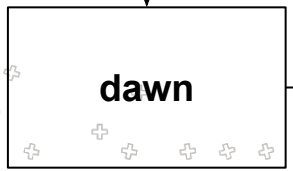
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



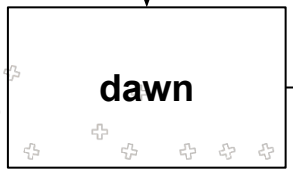

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```



MeteoSwiss

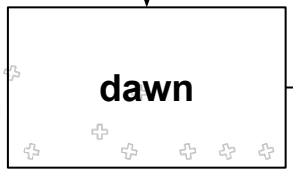
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!



MeteoSwiss

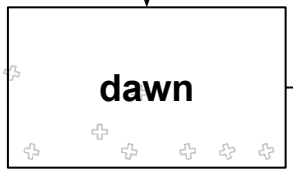
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!



MeteoSwiss

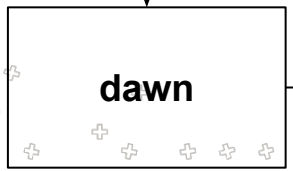
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify



MeteoSwiss

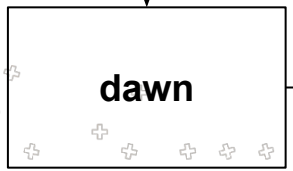
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField[Edge]
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify



MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(edgeIdx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF =
                eField[Edge>Cell>Edge]
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify

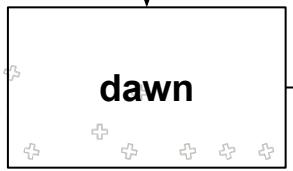
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(eIdxInner)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Same principle for reductions applies

```
@stencil
def mixed_fields(
   eIn: Field[Edge],
   eOut: Field[Edge],
   sparseF: Field[Edge > Cell > Edge]
):
   with levels_downward:
    eOut = sum_over(Edge > Cell > Edge,
                    sparseF*eIn[Edge])
```



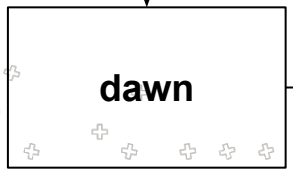
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        eOut(eIdx)+= eIn(eIdx)*sparseF(eIdx,linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Same principle for reductions applies

```
@stencil
def mixed_fields(
   eIn: Field[Edge],
   eOut: Field[Edge],
   sparseF: Field[Edge > Cell > Edge]
):
   with levels_downward:
    eOut = sum_over(Edge > Cell > Edge,
       sparseF*eIn[Edge > Cell > Edge])
```




```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        eOut(eIdx)+=
            eIn(eIdxInner)*sparseF(eIdx,linear_idx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these **horizontal offsets**

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF[Edge > Cell > Vertex] * eField[Edge])
```

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambiguous cases, user is *required* to state horizontal offsets

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField[Edge])
```

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField)
```

DuskSyntaxError: Field 'eField' requires a horizontal index inside of ambiguous neighbor iteration!

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets
- Horizontal offsets are *never allowed* on the left hand side of an assignment

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets
- Horizontal offsets are *never allowed* on the left hand side of an assignment

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut[Edge] = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField[Edge])
```

DuskSyntaxError: Invalid horizontal index for field 'eFieldOut' outside of neighbor iteration!

Nested Reductions

- Meaningful semantics can be assigned to nested reductions
- Early stage, not very well tested
- No code generation for cuda backend
  - No (apparent) use case in ICON dycore

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Outer Reduction

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Inner Reduction

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Inner Reduction
→ is executed |Edge > Cell| = 2 times

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Start of inner chain needs to match end of outer chain

Nested Reductions

Example

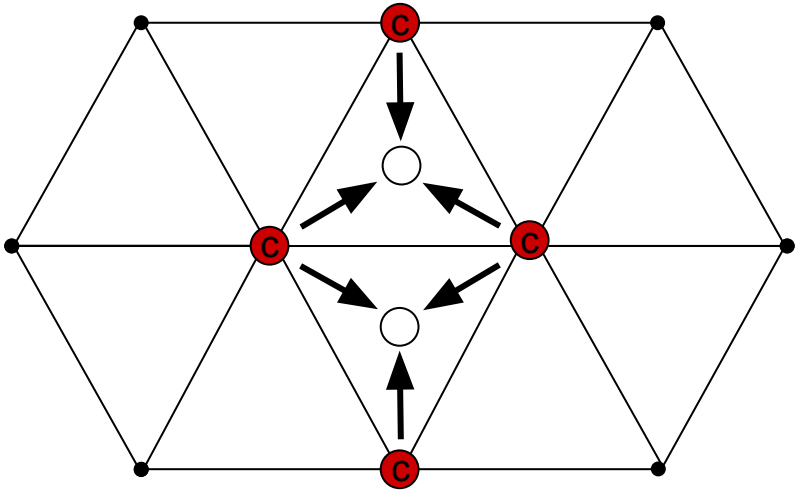
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Start of inner chain is return type of inner reduction
→ needs to form a (location) type consistent expression

Nested Reductions

Example

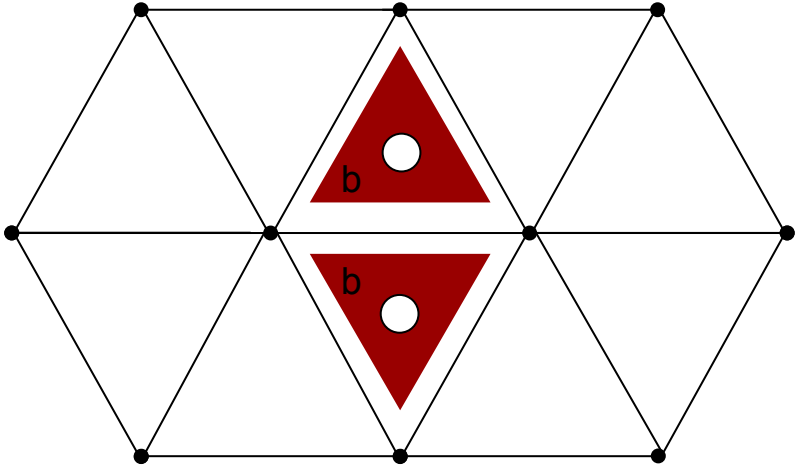
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

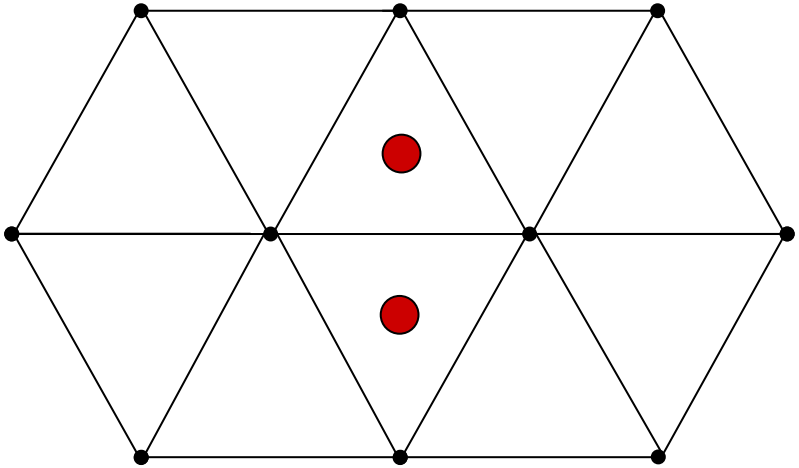
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

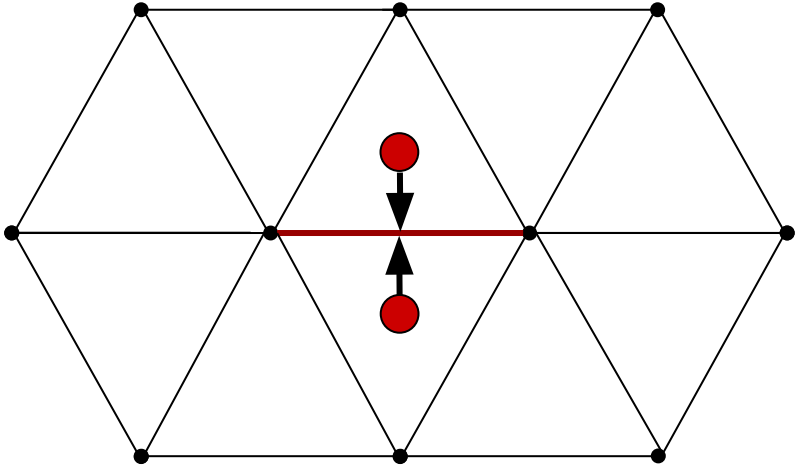
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

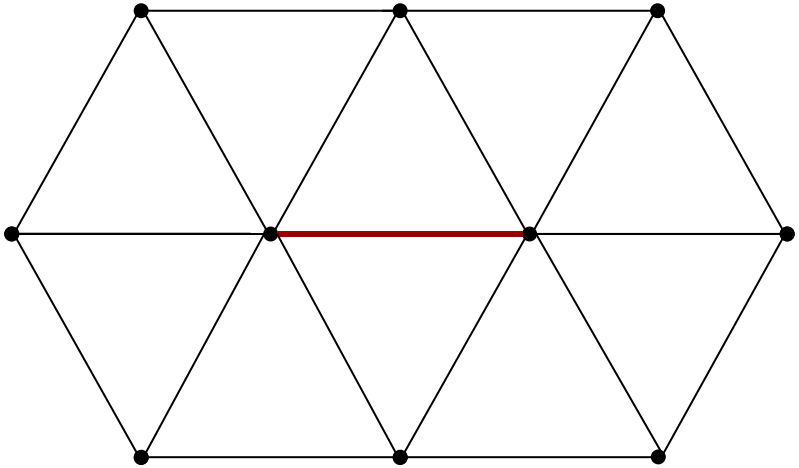
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions - Comparison to reduction with neighbor chain

Consider the following two examples:

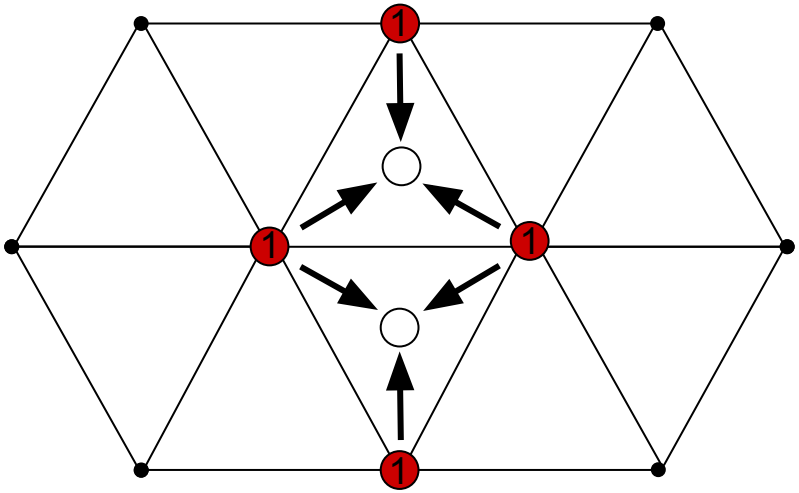
- are they different? equal?
- let's assume c(:) = 1. a(:) = ?

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```

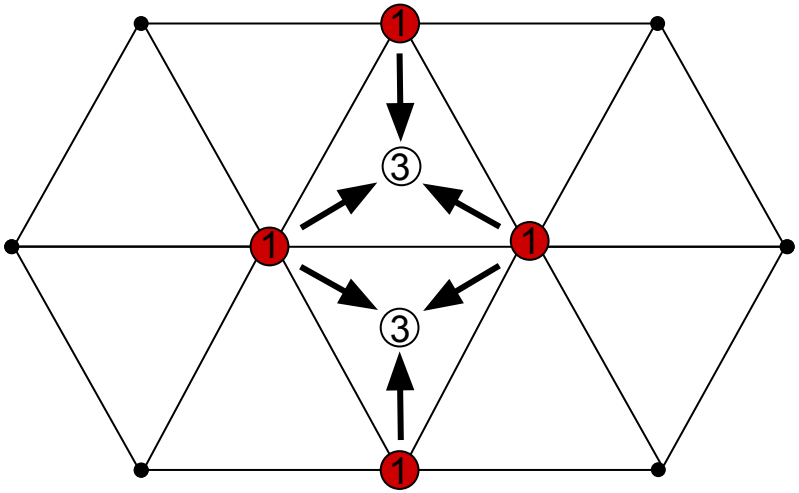
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



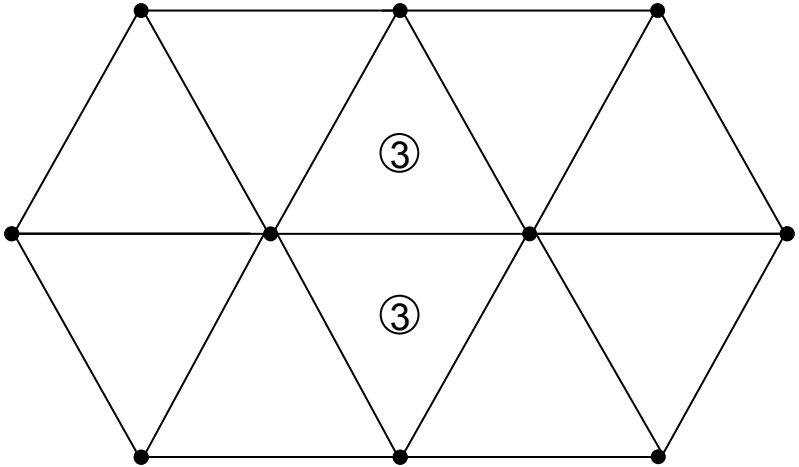
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



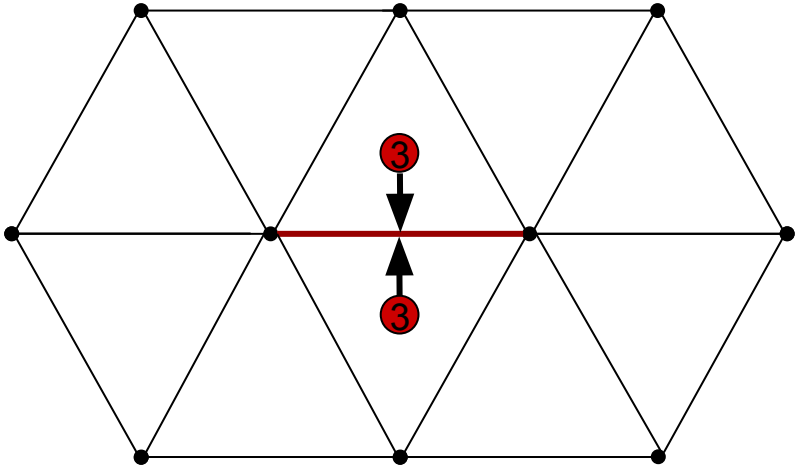
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



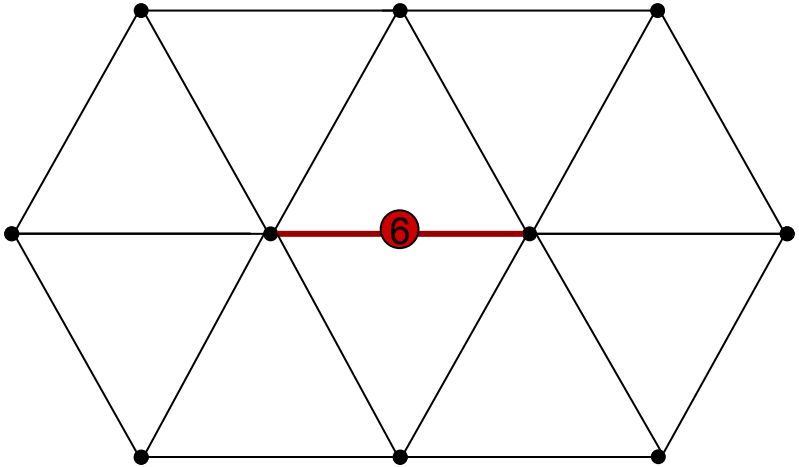
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



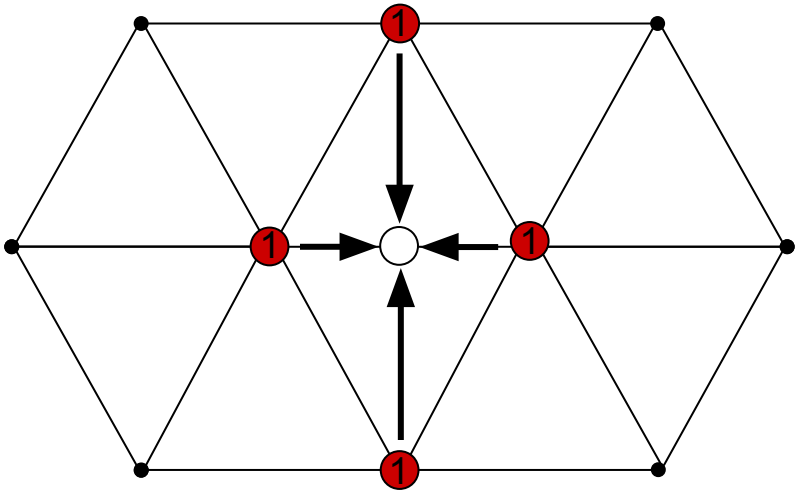
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



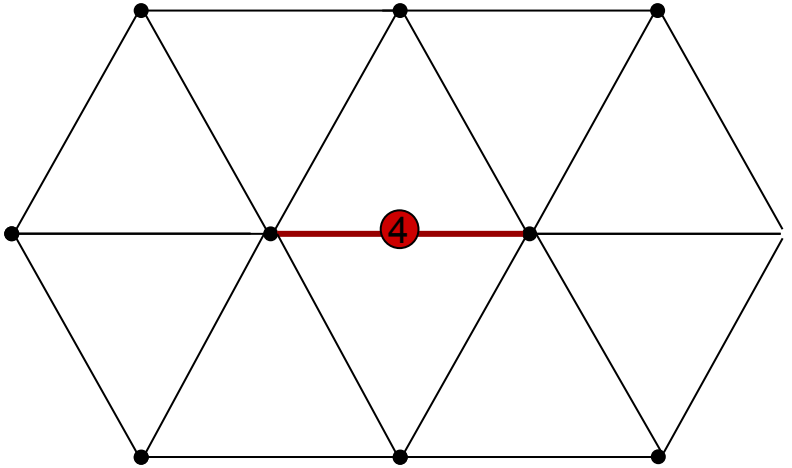
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```

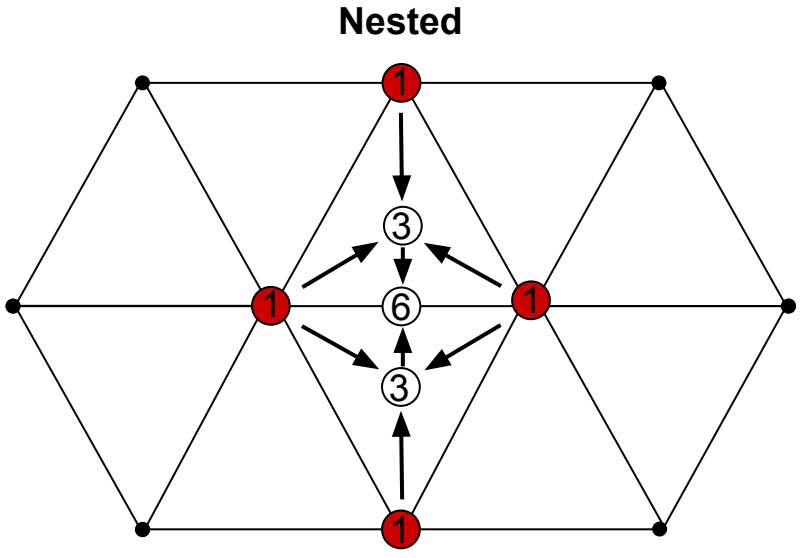


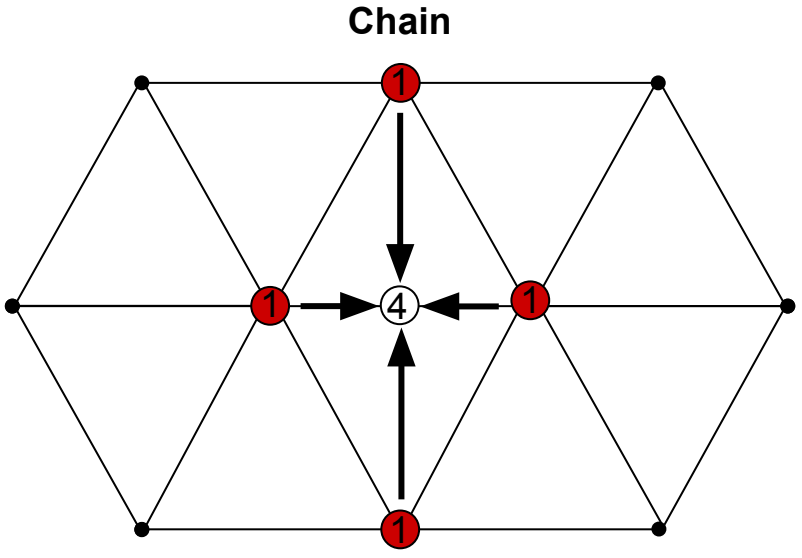
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```



Nested Reductions - Comparison to reduction with neighbor chain





Nested Reductions - Pushing the Envelope

Concepts are compatible / composable. Nothing stops you from...

- … nesting three levels (n levels) deep
- ... mixing nested with chained reductions
- … adding sparse fields

or even doing all of the above

Nested Reductions - Pushing the Envelope

Concepts are compatible / composable. Nothing stops you from...

- … nesting three levels (n levels) deep
- ... mixing nested with chained reductions
- … adding sparse fields

or even doing all of the above

```
@stencil
def wat(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex],
    sparseOuter: Field[Edge>Cell], sparseInner: Field[Vertex > Cell > Edge > Vertex]
):
    with levels_downward:
        a = sum_over(Edge > Cell,
                b*sparseOuter*sum_over(Cell > Vertex,
                    sum_over(Vertex > Cell > Edge > Vertex,
                        sparseInner[Vertex > Cell > Edge > Vertex]*c[Vertex > Cell > Edge > Vertex])))
```

Nested Reductions - Stencil Inlining

Stencil Inlining

```
@stencil
def reference(
field_a: Field[Vertex, K],
field_c: Field[Edge, K]
):
    field_b: Field[Cell, K]
    with levels_upward:
        field_b = sum_over(Cell > Vertex,
                           field_a)
        field_c = sum_over(Edge > Cell,
                           field_b)
```

inline

```
@stencil
def inlined(
field_a: Field[Vertex, K],
field_c: Field[Edge, K]
):
    with levels_upward:
        field_c = sum_over(Edge > Cell,
                           sum_over(Cell > Vertex, field_a))
```

Nested Reductions - Stencil Inlining

Stencil Inlining

- can be done automatically in an optimization pass
- has performance implications
- more on that later

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset. A quick reminder:

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset. A quick reminder:

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K]):
    with levels_upward as k:
        outF = inF[k+1]
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension

```
@stencil
def der_k(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[1:-1] as k:
        inF_k = (inF[k+1] - inF[k-1])/h[k]
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal

```
@stencil
def der_k(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[1:-1] as k:
        inF_k = (inF[k+1] - inF[k-1])/h[k]

@stencil
def f_k_high_order(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[0:-2] as k:
        inF_k = (inF[k+2] - 4*inF[k+1] + inF[k])/(2*h[k])
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal
- dusk & dawn is more flexible than that, the offset can also be read from a field instead:

```
@stencil
def indirect_read(inF: Field[Edge, K], outF: Field[Edge, K], indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection]
```

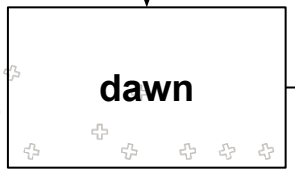
Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal
- dusk & dawn is more flexible than that, the offset can also be read from a field instead:
  - we can still combine this with a compile time literal offset

```
@stencil
def indirect_read(inF: Field[Edge, K], outF: Field[Edge, K], indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection+1]
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[k]
```

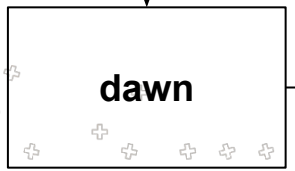


MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,k)
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection]
```

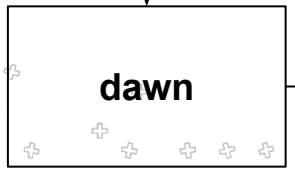


MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,indirection(eIdx,k))
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection+1]
```



MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,indirection(eIdx,k)+1)
```

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K]):
  with levels_upward as k:
      outF[k+1] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K], indir: IndexField[Edge, K]):
  with levels_upward as k:
      outF[indir] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K], indir: IndexField[Edge, K]):
  with levels_upward as k:
      indir = indir + 5
      outF[indir] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*
  - Need to either prepare them in driver code, or split stencil and switch type

```
@stencil
def prepare(v_vertical: Field[Edge, K], indir: Field[Edge, K], dt: Field[Edge, K]):
  with levels_upward as k:
      indir = -v_vertical * dt

@stencil
def use(indir: IndexField[Edge, K], ...):
  with levels_upward as k:
      ...
```

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*
  - Need to either prepare them in driver code, or split stencil and switch type

```
@stencil
def prepare(v_vertical: Field[Edge, K], indir: Field[Edge, K], dt: Field[Edge, K]):
  with levels_upward as k:
      indir = -v_vertical * dt

@stencil
def use(indir: IndexField[Edge, K], ...):
  with levels_upward as k:
      ...
```

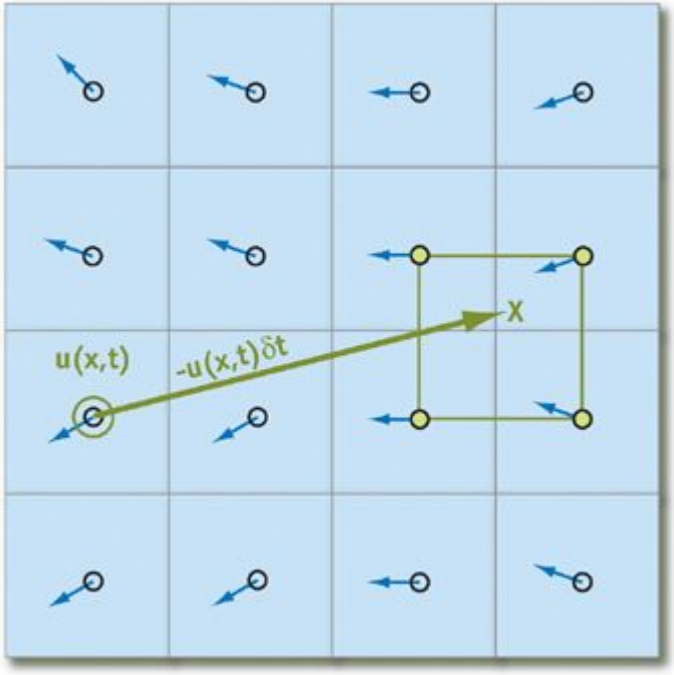
Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical

```
@stencil
def compute_btraj(v_vertical: Field[Cell, K], dt: Field[Cell,K], btraj: Field[Cell, K]):
  with levels_upward as k:
      btraj = -floor(v_vertical/dt)
```

Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical



taken from GPU Gems Chpt 38

Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical

```
@stencil
def advectT(T: Field[Cell,K], v_vertical: Field[Cell, K],
            dt: Field[Cell,K], btraj: IndexField[Cell, K],
            levels: Field[Cell, K], Tnew: Field[Cell,K]):
  # temperatures at corners of departure region
  T_depart_hi: Field[Cell, K]
  T_depart_lo: Field[Cell, K]
  # height coordinate corners of departure region
  h_depart_hi: Field[Cell, K]
  h_depart_lo: Field[Cell, K]
  # height coordinate at departure _point_
  h: Field[Cell, K]
  # lerp'ed Temp at departure point
  T_lerp: Field[Cell, K]
  with levels_upward as k:
      ...
```

Vertical Offsets & Indirections

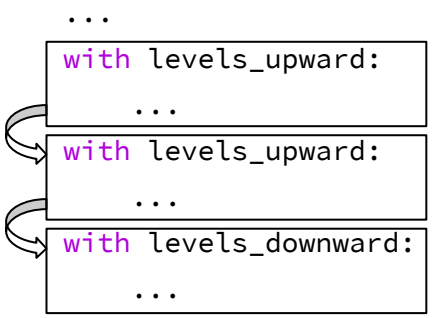
Typical Use Case: Semi-Lagrangian advection in the Vertical

```
...
with levels_upward as k:
    T_depart_hi = T[btraj+1]
    T_depart_lo = T[btraj]
    h_depart_hi = levels[btraj+1]
    h_depart_lo = levels[btraj]
    h = levels[k] - v_vertical*dt
    T_lerp = T_depart_lo +
               (h-h_depart_lo)*(T_depart_hi - T_depart_lo)/(h_depart_hi - h_depart_lo)
    Tnew = T_lerp
```

Parallel Model

- Let’s now see in detail how a dusk stencil is meant to be executed.
- The execution model (we also call it *parallel model*) presented here works as a contract with the (DSL) user.
- For each stencil, executing its generated code (which might be the result of various transformations and optimizations) *must* produce the same **effects** on output fields as the parallel model’s execution would when given the same inputs.
- The user thus doesn’t need to worry about what goes under the hood, as dusk&dawn promise that the generated code will behave equivalently to the parallel model’s execution.

Parallel Model



Looking at the code in a top-down fashion, the first “nodes” we encounter are *vertical domain regions*.

These constitute blocks of codes which must be executed **sequentially** in the order in which they appear.

Parallel Model

```
...
with levels_upward:
  STATEMENT1
  STATEMENT2
  STATEMENT3
    ...
```

Within each region:

- Iterate **sequentially** through the k-levels (upward or downward, depending on the `with levels_*` statement).
  - Within each k iteration execute the statements (direct children) of the region **sequentially** in the order in which they appear. The execution of a statement can start only when the preceding ones have completed.

Parallel Model

```
...
with levels_upward:
    f_a = f_b
    with sparse[Cell > Edge]:
        f_sparse = f_edge * f_cell
    if condition:
        f_c = 5.0
    ...
```

Each statement (direct child) of the region is executed on each location of the *horizontal* domain (location type depends on the statement) in **any order**.

This makes it an embarrassingly parallel formulation, allowing dawn to produce code that runs on several threads.

Parallel Model

```
...
with levels_upward:
    ...
    if f_c > 0.0:
        f_c = 0.0
    else:
        f_c = - f_c
    ...
```

This also means that statements containing substatements, such as if-then-else constructs and sparse loops, are to be considered atomic: they must be evaluated as a whole for each location.

Parallel Model

```
@stencil
def copy(
    inF: Field[Edge, K],
    inoutF: Field[Edge, K],
    outF: Field[Edge, K]
):

    tempF: Field[Edge, K]

    with levels_upward as k:
        tempF = inF
        outF = tempF
        inoutF = inoutF + tempF
```

API fields (part of the contract with the user):

- Input: must not be changed
- Output: doesn't matter what contained before, at the end of the stencil execution it must contain the correctly computed value
- Input-Output: same as output, but what contains at the beginning matters

Temporary fields (**not** part of the contract with the user):

Compiler has full leeway in what to do with them, e.g. keep or inline, …

The user should think of them as local "variables" with the scope of the stencil.

Parallel Model

```
@stencil
def reduction(
  f: Field[Edge, K]
):

  with levels_upward:
    f = sum_over(
          Edge > Cell > Edge,
          f[Edge > Cell > Edge]
        )
```

In the right hand sides of assignments, *value (copy) semantics* apply to fields being read.
The value is the **field as it was before the statement started being executed**.

Important point, will be clear why you need this later on...

Parallel Model: Execution Safety

We will now try to relax some constraints of the parallel model and highlight some criticalities that arise. This is to show what the compiler can/cannot do in order to optimize code.

Parallel Model: Execution Safety

```
@stencil
def vertical(
  f: Field[Edge, K],
  g: Field[Edge, K]
):
```

Remove sequentiality of **k-loop** iterations, allow **any order**. Opportunity to parallelize.

✖
```
with levels_upward as k:
  f = f + f[k-1]
```

✖
```
with levels_upward as k:
  g = g + 1.0
  f = g[k-1]
```

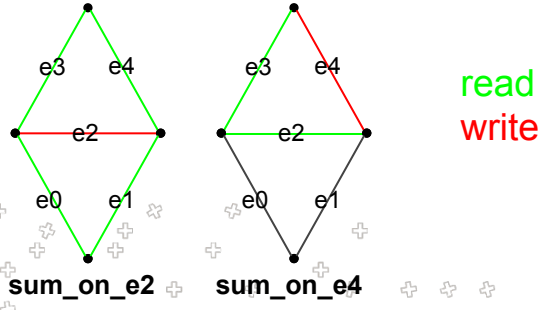
In general not possible when there are *vertical data dependencies* between statements (or of a statement with itself). But there are exceptions… (last slide)

✓
```
with levels_upward:
  f = g + 2.0
```

Any other case is ok.

Parallel Model: Execution Safety

```
@stencil
def reduction(
  f: Field[Edge, K]
):
  with levels_upward:
    f = sum_over(
          Edge > Cell > Edge,
          f[Edge > Cell > Edge])
```



Reference semantics on rhs instead of copy semantics. Accessing the actual field, as it is now.

This is **dangerous** because, depending on the order in which the statement is executed over the locations, the results change.

Think about swapping the order of sum_on_e2 and sum_on_e4.

An execution with multiple threads has exactly this kind of problem, which is called a **race condition**.

Copy semantics are the only safe option.

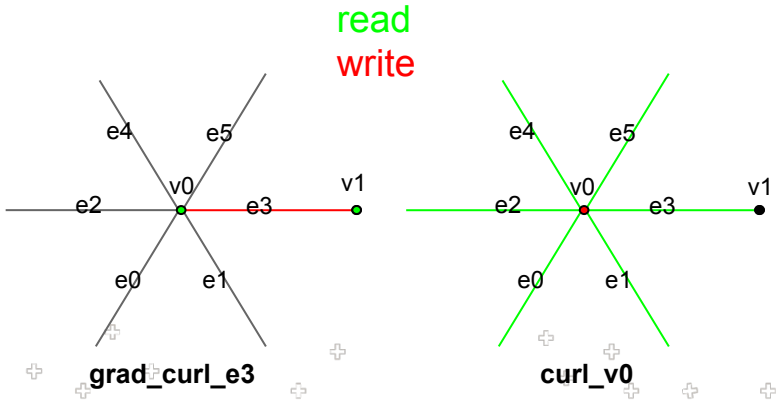
Parallel Model: Execution Safety

```
@stencil
def reduction(
  f: Field[Edge, K],
  grad_curl_f: Field[Edge, K]
):

  curl_f: Field[Edge, K]

  with levels_upward:
    curl_f =
       sum_over(Vertex > Edge,
                f * geofac_curl)
    grad_curl_f =
       sum_over(Edge > Vertex,
                curl_f,
                weights=[-1.0, 1])
```

One thread executing these statements for each location.

Remove **sequentiality of statements**.

For example, a threaded execution of statements within a k iteration doesn't guarantee the sequentiality of the statements inside.



Race condition!

Vertical Solvers

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T$$

To introduce vertical solvers, let's start from the *heat equation* that we have to solve.

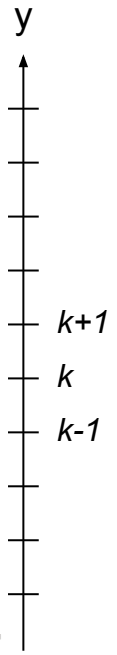
$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial y^2}$$

The focus here is to solve it along vertical columns of our domain, therefore we can directly look at the 1D heat equation.

Vertical Solvers

Usually in NWP, along the vertical, a fully **implicit discretization** scheme (backward Euler for time and second-order central finite difference for space) is employed (always numerically stable):

$$\frac{T_k^{n+1} - T_k^n}{\Delta t} = \kappa \frac{T_{k+1}^{n+1} - 2T_k^{n+1} + T_{k-1}^{n+1}}{(\Delta y)^2}$$



n: time point, *k*: vertical point

Vertical Solvers

Rearranging the recurrence equation:

$$T_k^{n+1} + \Delta t \kappa \frac{-T_{k+1}^{n+1} + 2T_k^{n+1} - T_{k-1}^{n+1}}{(\Delta y)^2} = T_k^n$$

In matrix form:

$$\left(\mathbb{I} + \frac{\Delta t \kappa}{(\Delta y)^2} \begin{bmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & -1 & 2 & \ddots & \\ & & \ddots & \ddots & -1 \\ 0 & & & -1 & 2 \end{bmatrix}\right) \begin{bmatrix} T_0^{n+1} \\ T_1^{n+1} \\ T_2^{n+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} T_0^n \\ T_1^n \\ T_2^n \\ \vdots \end{bmatrix}$$

Unknown

Vertical Solvers

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \ddots & \\ & & \ddots & \ddots & c_{i-2} \\ 0 & & & a_{i-1} & b_{i-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{i-1} \end{bmatrix}$$

A system of linear equations expressed through a tridiagonal matrix is solvable in linear time.

Vertical Solvers

```
@stencil
def TDMA(
 a: Field[Edge, K], b: Field[Edge, K], c: Field[Edge, K],
 d: Field[Edge, K], x: Field[Edge, K] ):

 g: Field[Edge, K]

 with levels_upward[0:0] as k:
   c = c / b
   d = d / b
 with levels_upward[1:] as k:
   g = 1.0 / (b - a * c[k-1])
   c = c * g
   d = (d - a * d[k-1]) * g
 with levels_downward[0:-1] as k:
   d -= c * d[k+1]
 with levels_upward:
   x = d
```

Forward sweep

Backward sweep

Thomas' algorithm to solve tridiagonal system of equations.

Applied column-wise over the whole domain.

Vertical Solvers

```
...
with levels_upward[1:] as k:
  f = f + f[k-1]
with levels_downward[0:-1] as k:
  g = g + g[k+1]
```

Solver-like access: vertically offset access to value written by previous iteration of k-loop.

If there is at least 1 solver-like access: **cannot parallelize k-loop**.

```
with levels_upward[0:-1] as k:
  f = f + f[k+1]
with levels_downward[1:] as k:
  g = g + g[k-1]
```

Stencil-like access: vertically offset access to value present before the k-loop.

If only stencil-like accesses: **can parallelize k-loop**.

Q&A

Questions?

Refresher

We have learned how to express basic stencil operators in dusk

```
@stencil
def grad_n(f_n: Field[Edge], dualL: Field[Edge], f: Field[Cell]):
    with levels_downward:
        f_n = sum_over(Edge > Cell, f, weights=[1,-1]) / dualL
```

```
@stencil
def divergence(vn: Field[Edge], L: Field[Edge], A: Field[Cell], edge_orientation:
Field[Cell > Edge], div: Field[Cell]):
  with levels_downward:
      div = sum_over(Cell > Edge, vn * L * edge_orientation) / A
```

Combining operators

An typical PDE operator needs to be expressed as a combination of various basic stencil operators.

E.g. the FVM vector laplacian:

$$\nabla^2 \mathbf{v} = \nabla(\nabla \cdot \mathbf{v}) - \nabla \times (\nabla \times \mathbf{v})$$

In its discretized form:

$$\langle \nabla^2 \mathbf{v} \rangle_{\mathbf{e}} = \langle \nabla \langle \nabla \cdot \mathbf{v} \rangle_{\mathbf{c}} \rangle_{\mathbf{e}} - \langle \nabla \times \langle \mathbf{v} \rangle_{\mathbf{v}} \rangle_{\mathbf{e}}$$