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 - Basic Concepts

Basic Operations on Unstructured Meshes

Basic Concepts

Overview:

- Vertical Looping
  - Execution Safety
- Type Consistency
- Reductions
- Conditionals

Vertical Looping

Structure of a dusk program

```
@stencil
def copy_on_vertex(input: Field[Vertex,K], output: Field[Vertex,K]):
    with levels_upward:
        output = input
```

Signature

MeteoSwiss

Vertical Looping

Structure of a dusk program

```
@stencil
def copy_on_vertex(input: Field[Vertex,K], output: Field[Vertex,K]):
    with levels_upward:
        output = input
```

} Vertical Domain / Loop Order

MeteoSwiss

Vertical Looping

Structure of a dusk program

```
@stencil
def copy_on_vertex(input: Field[Vertex,K], output: Field[Vertex,K]):
    with levels_upward:
        output = input
```

} List of Statements

Vertical Looping

Structure of a dusk program

```
@stencil
def copy_on_vertex(input: Field[Vertex,K], output: Field[Vertex,K]):
    with levels_upward:
        output = input
```

Loop over the vertical domain (explicit)

- loops over the columns
- dawn **tries to** emit parallel code for this loop

Loop over the horizontal domain (implicit)

- Loops over Vertices in this case (since both fields are on vertices)
- **Always** run in parallel

Vertical Looping

A closer look at the `with levels_*` statement

- every statement needs to be contained in a `with levels_*` statement
- `with levels_*` statements may not be nested

```
@stencil
def copy_on_vertex(...):
    with levels_upward:
        output = input
```

MeteoSwiss

Vertical Looping

A closer look at the `with levels_*` statement

- every statement needs to be contained in a `with levels_*` statement
- `with levels_*` statements may not be nested
- the user may choose between `levels_upward` and `levels_downward`, to indicate a loop starting either from the lowest or or highest vertical level

```
@stencil
def copy_on_vertex(...):
    with levels_upward:
        output = input
    with levels_downward:
        output = input
```

Vertical Looping

A closer look at the `with levels_*` statement

- every statement needs to be contained in a `with levels_*` statement
- `with levels_*` statements may not be nested
- the user may choose between `levels_upward` and `levels_downward`, to indicate a loop starting either from the lowest or or highest vertical level
- The iteration variable may be accessed by giving it a name, e.g. k
  - This can be used to read with an offset

```
@stencil
def copy_on_vertex(...):
    with levels_upward as k:
        output = input[k+1]
```

MeteoSwiss

Vertical Looping

A closer look at the `with levels_*` statement

- every statement needs to be contained in a `with levels_*` statement
- `with levels_*` statements may not be nested
- the user may choose between `levels_upward` and `levels_downward`, to indicate a loop starting either from the lowest or or highest vertical level
- The iteration variable may be accessed by giving it a name, e.g. k
  - This can be used to read with an offset
  - Offset writes are prohibited!

Illegal Code!

```
@stencil
def copy_on_vertex(...):
   with levels_upward as k:
       output[k+1] = input
```

Vertical Looping

A closer look at the `with levels_*` statement

- every statement needs to be contained in a `with levels_*` statement
- `with levels_*` statements may not be nested
- the user may choose between `levels_upward` and `levels_downward`, to indicate a loop starting either from the lowest or or highest vertical level
- The iteration variable may be accessed by giving it a name, e.g. k
  - This can be used to read with an offset
  - Offset writes are prohibited!
- You can iterate on a slice of the vertical dimensions only
  - The example on the right hand side would iterate from the fifth level up to five levels from the top

```
@stencil
def copy_on_vertex(...):
    with levels_upward[5:-5] as k:
        output = input
```

Vertical Looping

dusk code

```
@stencil
def copy_on_vertex(input: Field[Vertex,K],
                   output: Field[Vertex,K]):
   with levels_upward:
       output = input
```

dawn

serial pseudo code

```
for (k = 0; k < kmax; k++)
   for (vIdx = 0; vIdx < mesh.num_vertices(); vIdx++)
       output(vIdx, k) = input(vIdx, k)
```

Vertical Looping

dusk code

```
@stencil
def copy_on_vertex(input: Field[Vertex,K],
                   output: Field[Vertex,K]):
    with levels_upward:
        output = input
```

parallel pseudo code

```
parfor (k = 0; k < kmax; k++)
    parfor (vIdx = 0; vIdx < mesh.num_vertices(); vIdx++)
        output(vIdx, k) = input(vIdx, k)
```

dawn

Vertical Looping

dusk code

```
@stencil
def copy_on_vertex(input: Field[Vertex,K],
                   output: Field[Vertex,K]):
  with levels_upward:
     output = input
```

parallel pseudo code

```
parfor (k = 0; k < kmax; k++)
  parfor (vIdx = 0; vIdx < mesh.num_vertices(); vIdx++)
     output(vIdx, k) = input(vIdx, k)
```

dawn

Vertical Looping - Parallelization

- dawn will always **try to** emit parallel code for the vertical
- there are certain situations where this is not possible
  - i.e. the code written necessitates serial execution of the vertical loop
  - this happens for certain patterns of vertical offset reads
- For now assume that parallelization is always possible
  - whether dusk program says `levels_downward` or `levels_upward` is of no consequence (for now)
  - you can safely assume that all exercises don't exhibit such patterns, you don't need to touch the vertical iteration direction in any of them

Vertical Looping - Parallelization - Safety

Let's look at a pseudo code example:

```
for (k = 0; k < kmax-1; k++)
   for (cellIdx = 0; cellIdx < mesh.num_cells(); cellIdx++)
      inout(cellIdx, k) = inout(cellIdx, k+1)
```

So essentially you would like to shift each value one level downward along the vertical axis

Vertical Looping - Parallelization - Safety

Later you decide to parallelize this snippet. You come up with:

```
parfor (k = 0; k < kmax-1; k++)
   parfor (cellIdx = 0; cellIdx < mesh.num_cells(); cellIdx++)
       inout(cellIdx, k) = inout(cellIdx, k+1)
```

Vertical Looping - Parallelization - Safety

Later you decide to parallelize this snippet. You come up with:

```
parfor (k = 0; k < kmax-1; k++)
   parfor (cellIdx = 0; cellIdx < mesh.num_cells(); cellIdx++)
       inout(cellIdx, k) = inout(cellIdx, k+1)
```

- This is a race condition!
- Depending on whether `inout(cellIdx, k+1)` has already been written to by another thread, the result will differ!

Vertical Looping - Parallelization - Safety

Later you decide to parallelize this snippet. You come up with:

```
parfor (k = 0; k < kmax-1; k++)
  parfor (cellIdx = 0; cellIdx < mesh.num_cells(); cellIdx++)
      inout(cellIdx, k) = inout(cellIdx, k+1)
```

DANGEROUS CODE

- This is a race condition!
- Depending on whether `inout(cellIdx, k+1)` has already been written to by another thread, the result will differ!

Vertical Looping - Parallelization - Safety

Lets try the same thing again in dawn:

```
@stencil
def shift(inout: Field[Cells,K]):
   with levels_upward as k:
       inout = inout[k+1]
```

- dawn is a parallelizing compiler. It knows about parallelization and its perils
- so we would either expect dawn to
  - reject this code
  - emit a stern warning that this is unsafe
  - transform the code to be safe somehow
  - …?
- let's see what happens!

Vertical Looping - Parallelization - Safety

dusk code

```
@stencil
def shift(inout: Field[Cells,K]):
    with levels_upward as k:
        inout = inout[k+1]
```

dawn

parallel pseudo code

```
inout_0 = new cell_field(...)
parfor (k = 0; k < kmax; k++)
  parfor (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
      inout_0(cIdx, k) = inout(cIdx, k)
sync() // wait for all threads
parfor (k = 0; k < kmax-1; k++)
  parfor (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
      inout(cIdx, k) = inout_0(cIdx, k+1)
```

Vertical Looping - Parallelization - Safety

So in summary

- dawn noticed the data dependency
- made a temporary copy of the input field
  - this is called *field versioning*
- ensured that versioning the field was run in parallel
- and finally ran the shift safely in parallel

→ This is one of many situations where dawn emits correct code automatically that would require re-engineering to run in parallel using conventional compilers

Type System & Type Checking

- As discussed, in Finite Volume Codes each variable is either located on a **Cell**, a **Vertex** or an **Edge**.
- This fact is directly reflected in the dusk & dawn type system
- Any field may have a horizontal dimension, vertical dimension, or both

Actually, all **simple** types (more complex ones later) in dusk / dawn fit on this slide:

- Horizontal Field Types

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

- The Vertical Field Type

```
vertField: Field[K]
```

- "Full" Fields (Both Horizontal and Vertical Dimension)

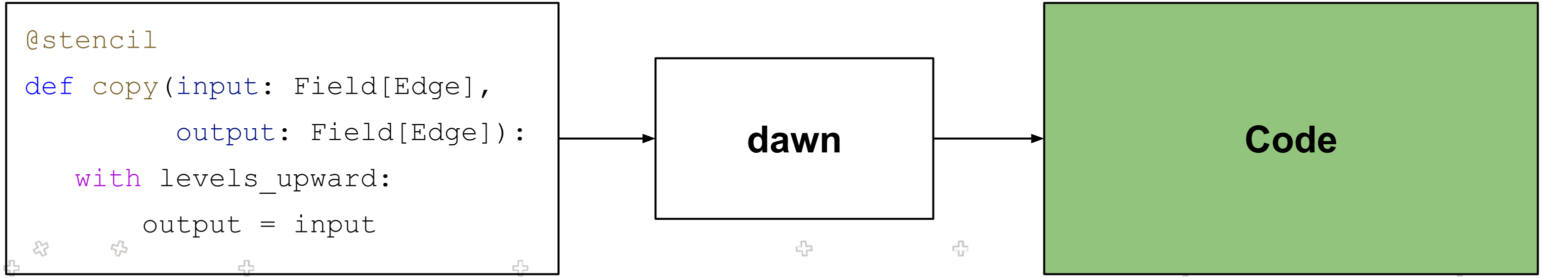
```
vField3D: Field[Vertex,K], eField3D: Field[Edge,K], cField3D: Field[Cell,K]
```

Type System & Type Checking

- What about the individual entries of the fields?
  - what is stored e.g. for each edge in a `eField: Field[Edge]`
- Currently, dawn only supports float, either in 32 or 64 bit precision
  - controlled by a flag in driver code
- In the future, we want to support more primitive types (int, bool, …) as well as more complex types such as (2d/3d) vectors
  - for now, emulate vector fields using two (three) individual fields
    `vx: Field[Edge], vy: Field[Edge], (vz: Field[Edge])`

Type System & Type Checking

- In summary, dusk & dawn types consist of
  - dimensionality
  - location
- dawn implements strict type checking to avoid errors
- in binary operations and assignments, the **location** of the left hand side needs to match the location on the right hand side:



Type System & Type Checking

- In summary, dusk & dawn types consist of
  - dimensionality
  - location
- dawn implements strict type checking to avoid errors
- in binary operations and assignments, the **location** of the left hand side needs to match the location on the right hand side:

```
@stencil
def copy(input: Field[Edge],
         output: Field[Cell]):
    with levels_upward:
        output = input
```

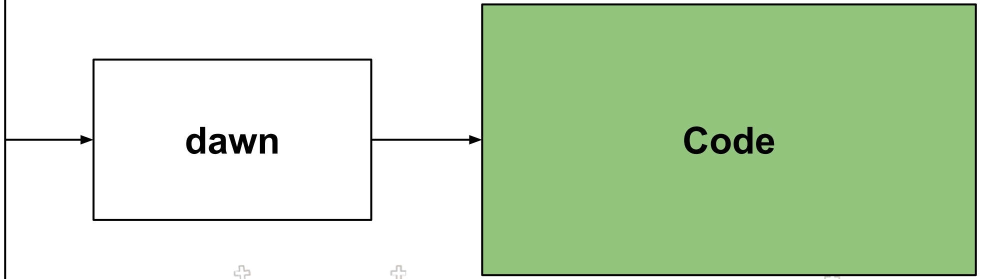
→ **dawn** →

Type Error:
Assignment at line...

Type System & Type Checking

- In summary, dusk & dawn types consist of
  - dimensionality
  - location
- dawn implements strict type checking to avoid errors
- in binary operations and assignments, the **location** of the left hand side needs to match the location on the right hand side:

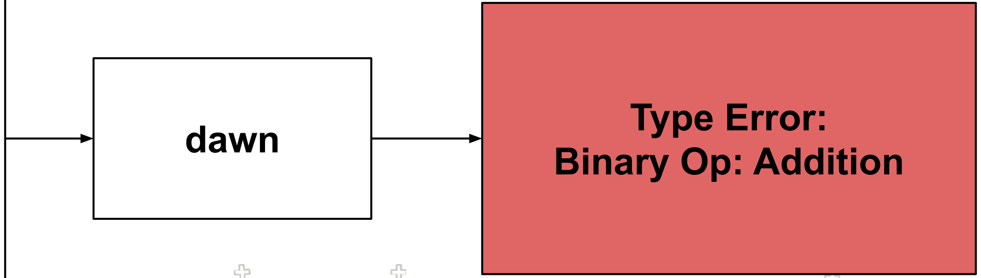
```
@stencil
def add(lhs: Field[Edge],
        a: Field[Edge],
        b: Field[Edge]):
    with levels_upward:
        lhs = a + b
```



Type System & Type Checking

- In summary, dusk & dawn types consist of
  - dimensionality
  - location
- dawn implements strict type checking to avoid errors
- in binary operations and assignments, the **location** of the left hand side needs to match the location on the right hand side:

```
@stencil
def add(lhs: Field[Edge],
        a: Field[Edge],
        b: Field[Cell]):
    with levels_upward:
        lhs = a + b
```



Type System & Type Checking

- It's quite simple to ensure the same level of safety in any modern programming language
- However, model code is often written in unsafe manners, e.g.

```
double* lhs = new double[mesh.num_edges()];
double* a = new double[mesh.num_edges()];
double* b = new double[mesh.num_cells()];
for (int eIdx = 0; eIdx++ < mesh.num_edges(); eIdx++) {
      lhs[eIdx] = a[eIdx] + b[eIdx];
}
```

- Would compile with no type error
- Would segfault (in the best case)
- Overwrite some other memory (in the worst case)
- (Types are checked at compile time, hence has no runtime impact)

Type System & Type Checking

- It's quite simple to ensure the same level of safety in any modern programming language
- Sketch of safe version

```
edge_field* lhs = new edge_field(mesh.num_edges());
edge_field* a = new edge_field(mesh.num_edges());
cell_field* b = new cell_field(mesh.num_cells());
for (edge_iter eIter = mesh.edges().begin(); eIter != mesh.edges().end() ; eIter++) {
      lhs->at(eIter) = a->at(eIter) + b->at(eIter); //COMPILER ERROR!
}
```

Type System & Type Checking

We talked about **location**, what about **dimensionality**?

- For Assignments, consider the following table:

lhs\rhs	1D	2D	3D
3D			
2D			
1D			

Type System & Type Checking

We talked about **location**, what about **dimensionality**?

- For Assignments, consider the following table:

lhs\rhs	1D	2D	3D
3D			
2D			
1D			

1D = "vertical"
2D = "horizontal"

```
double* vert = new double[num_k];
double* hCells = new double[mesh.num_cells()];
```

Type System & Type Checking

We talked about **location**, what about **dimensionality**?

- For Assignments, consider the following table:

lhs\rhs	1D	2D	3D
3D			
2D			
1D			

```
for (k = 0; k < kmax; k++)
  for (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
    cField3D(cIdx, k) = cField2D(cIdx)
```

```
for (k = 0; k < kmax; k++)
  for (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
    cField3D(cIdx, k) = cField1D(k)
```

Type System & Type Checking

We talked about **location**, what about **dimensionality**?

- For Assignments, consider the following table
- For Binary Operations all combinations are ok

Type System & Type Checking

We talked about **location**, what about **dimensionality**?

- For Assignments, consider the following table
- For Binary Operations all combinations are ok

```
@stencil
def dimensions(f3d: Field[Vertex,K],
               f2d: Field[Vertex], f1d: Field[K]):
   with levels_upward:
       f3d = f2d + f1d
```

dawn

```
for (k = 0; k < kmax; k++)
  for (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
      cField3D(cIdx, k) = cField2D(cidx) + cField1D(k)
```

Quick Recap

So what can we do so far?

- We can copy fields around
  - with vertical offset if desired
- We can do arithmetic on fields

… As long as the fields involved are all on the same location

Q&A

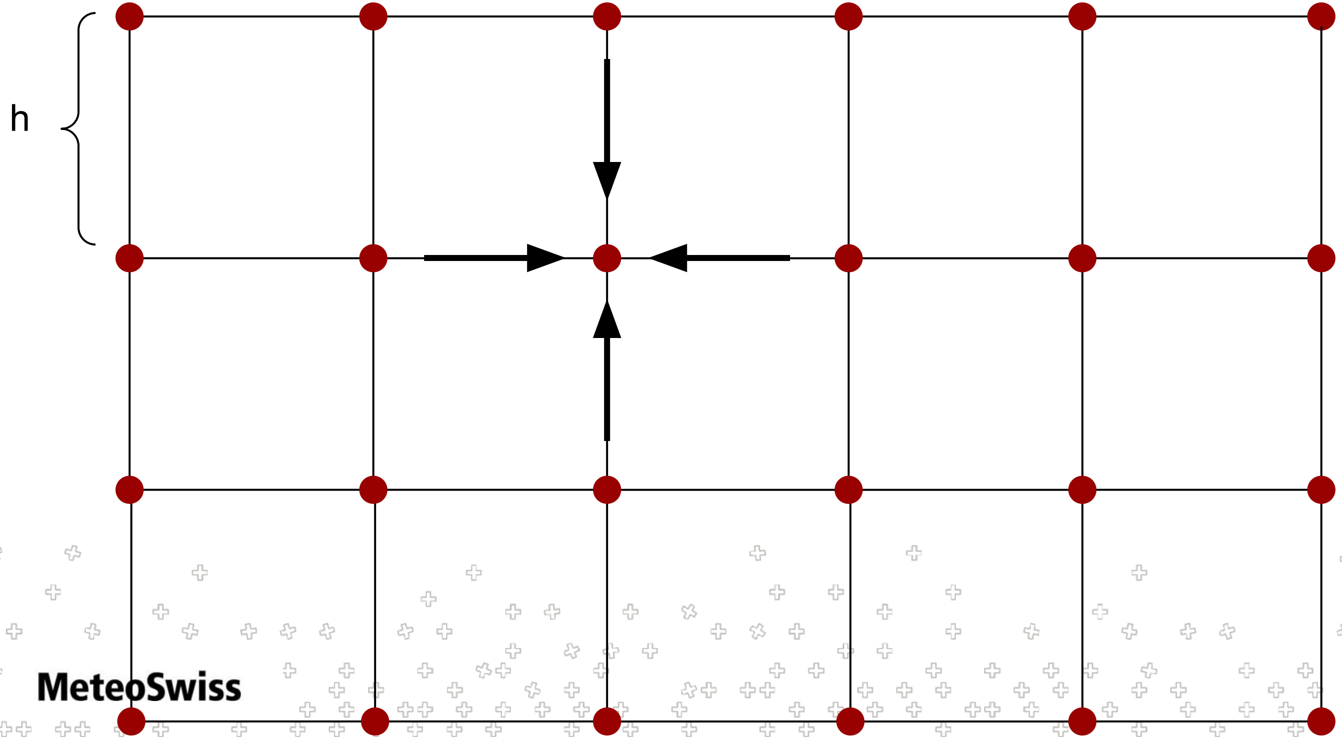
Questions?

Compact Stencil

- The compact stencil is the basic numerical concept supported
- Roughly: "algebraic combination of values located at a central point and values located at adjacent points"
- Possibly most well known from Finite Differences

Compact Stencil

$$\nabla^2 f(i,j) = \frac{f(i+1,j) + f(i,j+1) - 4f(i,j) + f(i-1,j) + f(i,j-1)}{h^2}$$



Compact Stencil

- The compact stencil is the basic numerical concept supported
- Roughly: "algebraic combination of values located at a central point and values located at **adjacent points"**
- Possibly most well known from Finite Differences
- On a Cartesian mesh the adjacent points can easily be addressed as we just have seen

$$\nabla^2 f(i,j) = \frac{f(\mathbf{i+1,j}) + f(\mathbf{i,j+1}) - 4f(\mathbf{i,j}) + f(\mathbf{i-1,j}) + f(\mathbf{i,j-1})}{h^2}$$

- Not true on more general (FVM) Meshes

Intermission - The Most Basic FVM Computation

Consider a Conservation law

$$\frac{\partial}{\partial t}u(x,t) + \nabla \cdot f(u(x,t)) = \underbrace{g(u(x,t))}_{\text{source terms}} \rightarrow 0$$

Assume u is constant over a small control volumes Ω_i (u(t) → u in the following for legibility)

$$\int_{\Omega_i} \frac{\partial}{\partial t} u \, d\Omega_i + \int_{\Omega_i} \nabla \cdot f(u) \, d\Omega = 0$$

∇ of unknown quantity f → apply divergence theorem

$$\int_{\Omega_i} \frac{\partial}{\partial t} u \, d\Omega_i + \int_{\delta\Omega_i} f(u) \cdot n \, dS = 0$$

Intermission - The Most Basic FVM Computation

∇ of unknown quantity f → apply divergence theorem

$$\int_{\Omega_i} \frac{\partial}{\partial t} u \, d\Omega_i + \int_{\delta\Omega_i} f(u) \cdot n \, dS = 0$$

a few more basic manipulations

$$\frac{\partial}{\partial t} u \underbrace{\int_{\Omega_i} d\Omega_i}_{|\Omega_i|} + \int_{\delta\Omega_i} f(u) \cdot n \, dS = 0$$

$$\frac{\partial}{\partial t} u = -\frac{1}{|\Omega_i|} \int_{\delta\Omega_i} f(u) \cdot n \, dS$$

Intermission - The Most Basic FVM Computation

$$\frac{\partial}{\partial t} u = -\frac{1}{|\Omega_i|} \int_{\delta\Omega_i} f(u) \cdot n \, dS$$

Discretize on a Finite Volume Mesh (e.g. triangular)

$$\underbrace{\frac{\partial}{\partial t} u(\Omega_i) = -\frac{1}{|\Omega_i|}}_{\text{Cell Quantities}} \sum_{e=1}^{3} \underbrace{f(u)_e \cdot n_e L_e}_{\text{Edge Quantities}}$$

Intermission - The Most Basic FVM Computation

$$\frac{\partial}{\partial t} u = -\frac{1}{|\Omega_i|} \int_{\delta\Omega_i} f(u) \cdot n \, dS$$

Discretize on a Finite Volume Mesh (e.g. triangular)

$$\underbrace{\frac{\partial}{\partial t} u(\Omega_i) = -\frac{1}{|\Omega_i|}}_{\text{Cell Quantities}} \underbrace{\sum_{e=1}^{3}}_{} \underbrace{f(u)_e \cdot n_e L_e}_{\text{Edge Quantities}}$$

Cell Quantities

Edge Quantities

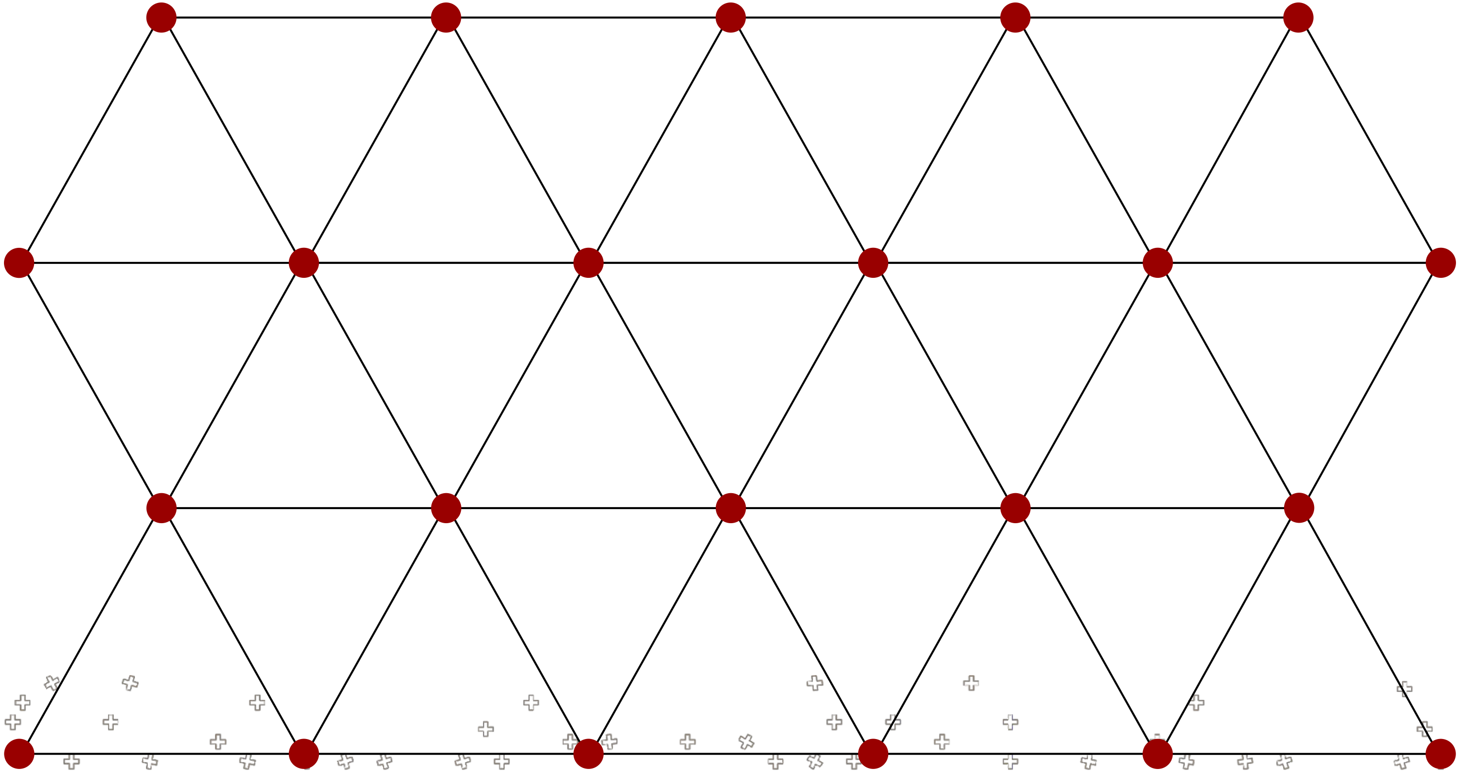
sum, but in more general terms, a **reduction**

Reductions

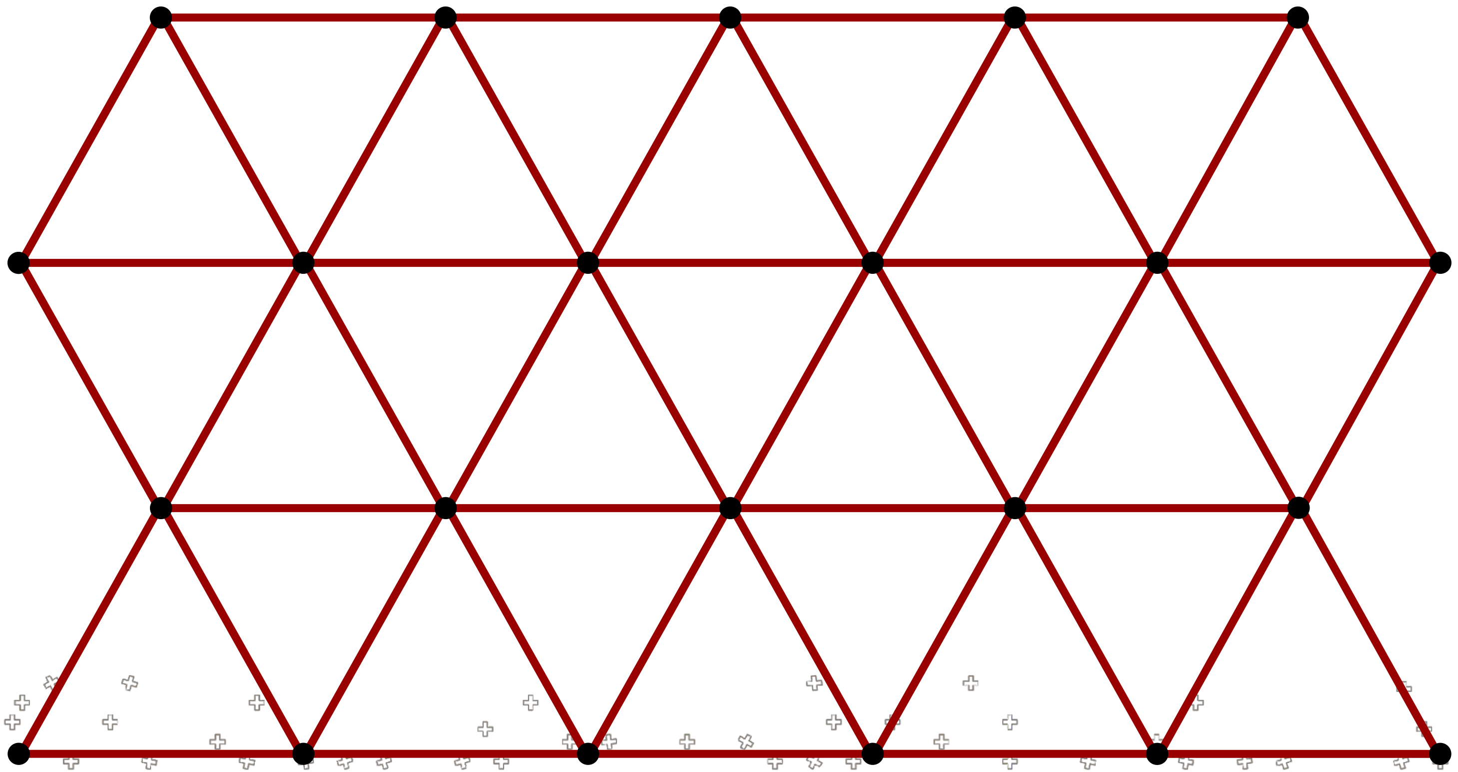
- Reductions are to FVM what stencils are to FD
- One of the most important, if not *the* most important, primitive in dawn
- Implemented as general as possible
  - Stated goal: be able to map every FORTRAN reduction in the ICON dycore to dusk & dawn reductions

- Reductions are closely linked to the concept of neighborhoods on unstructured / FVM meshes

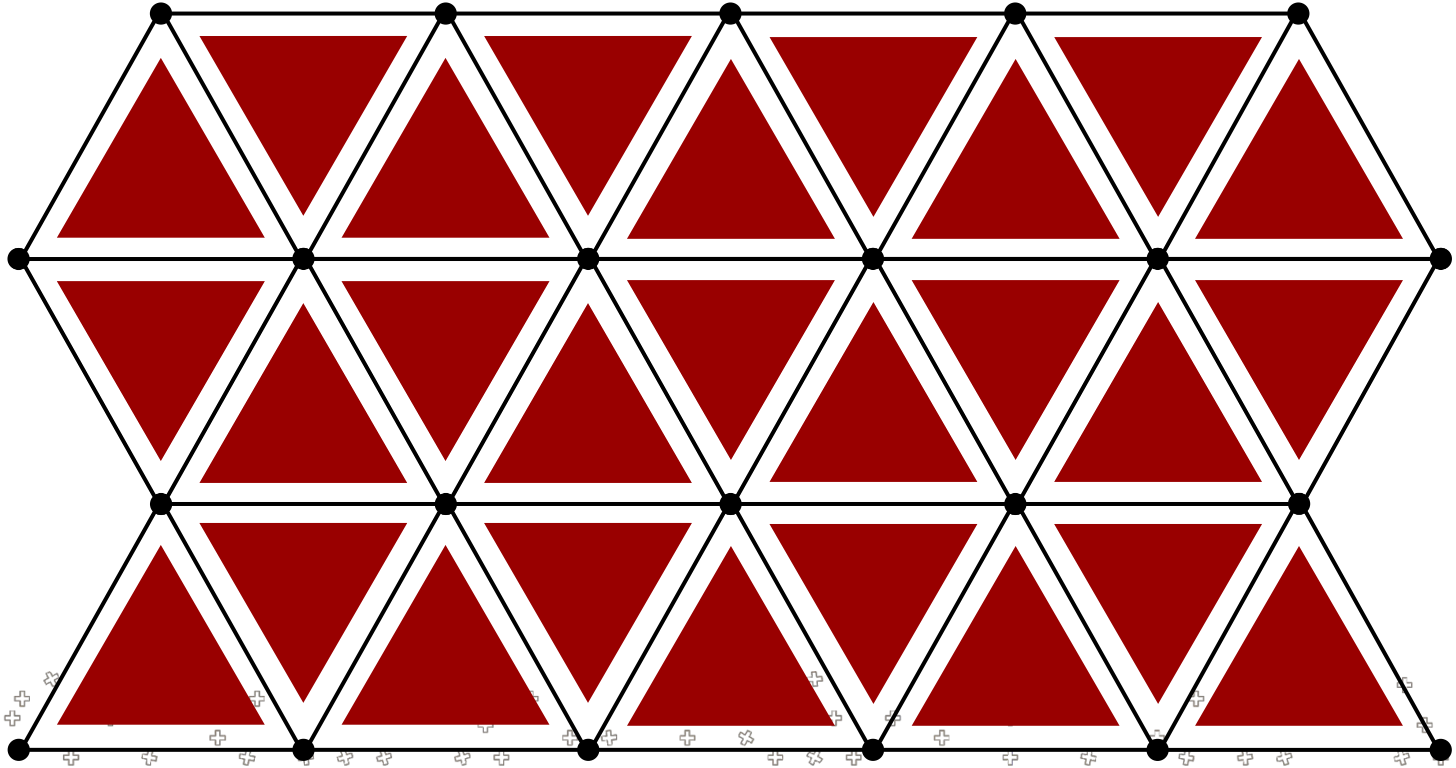
Mesh: Vertices



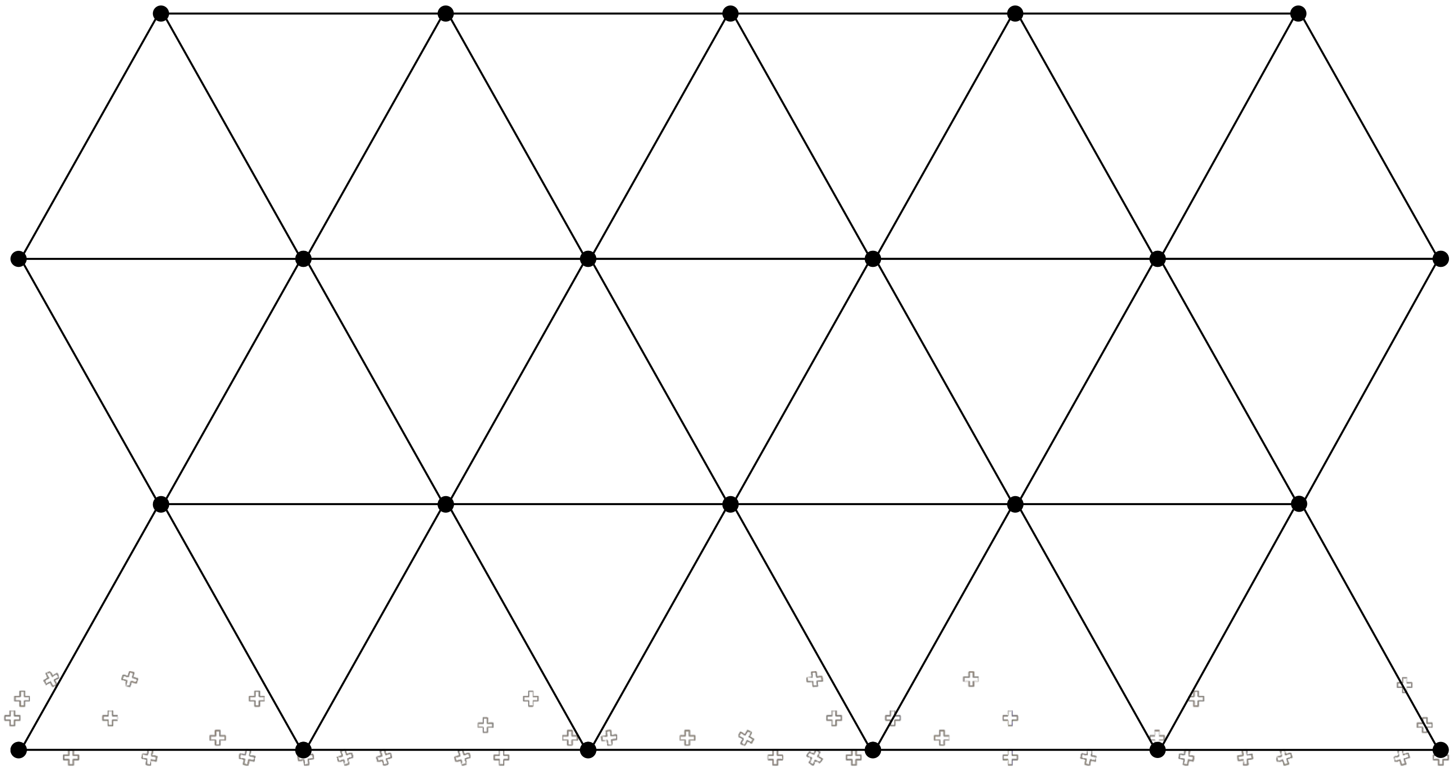
Mesh: Edges



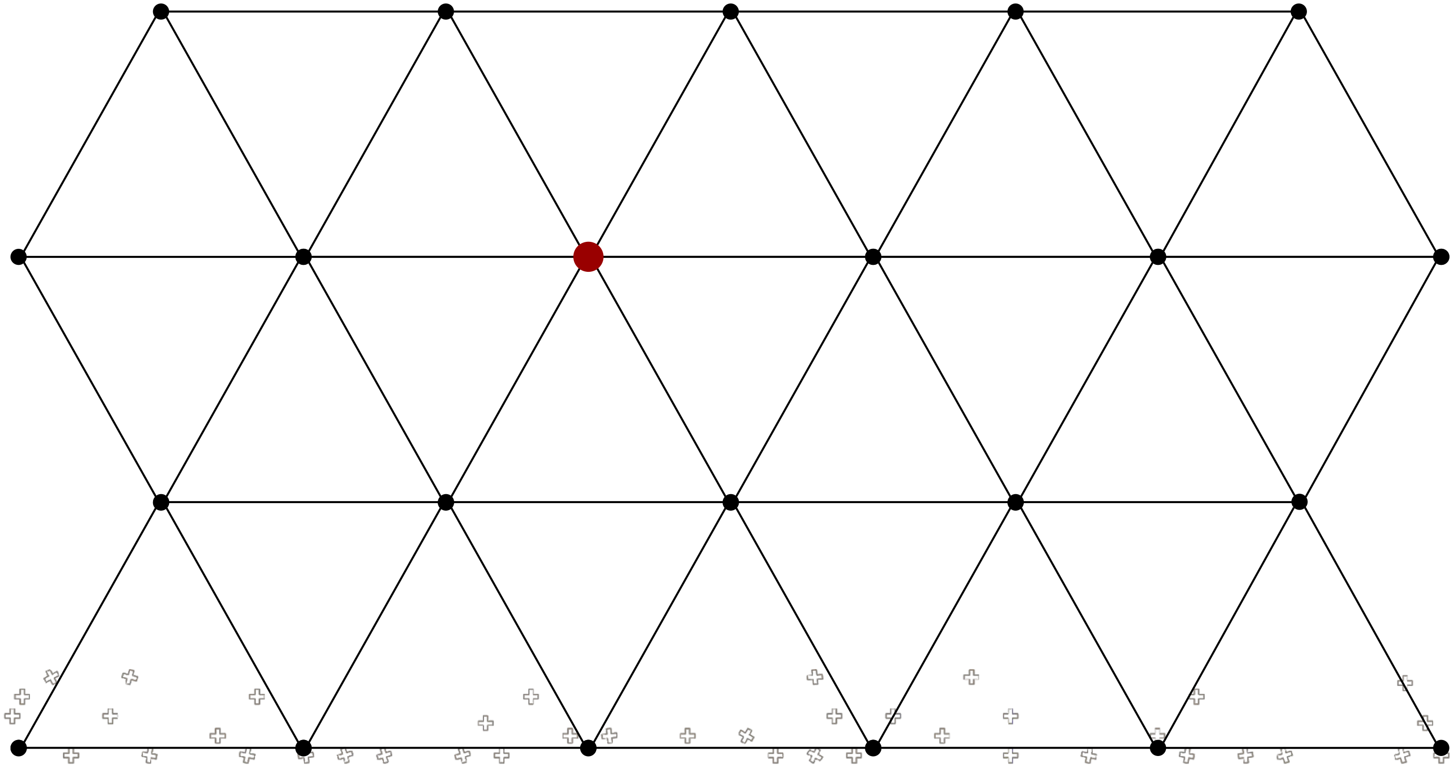
Mesh: Cells



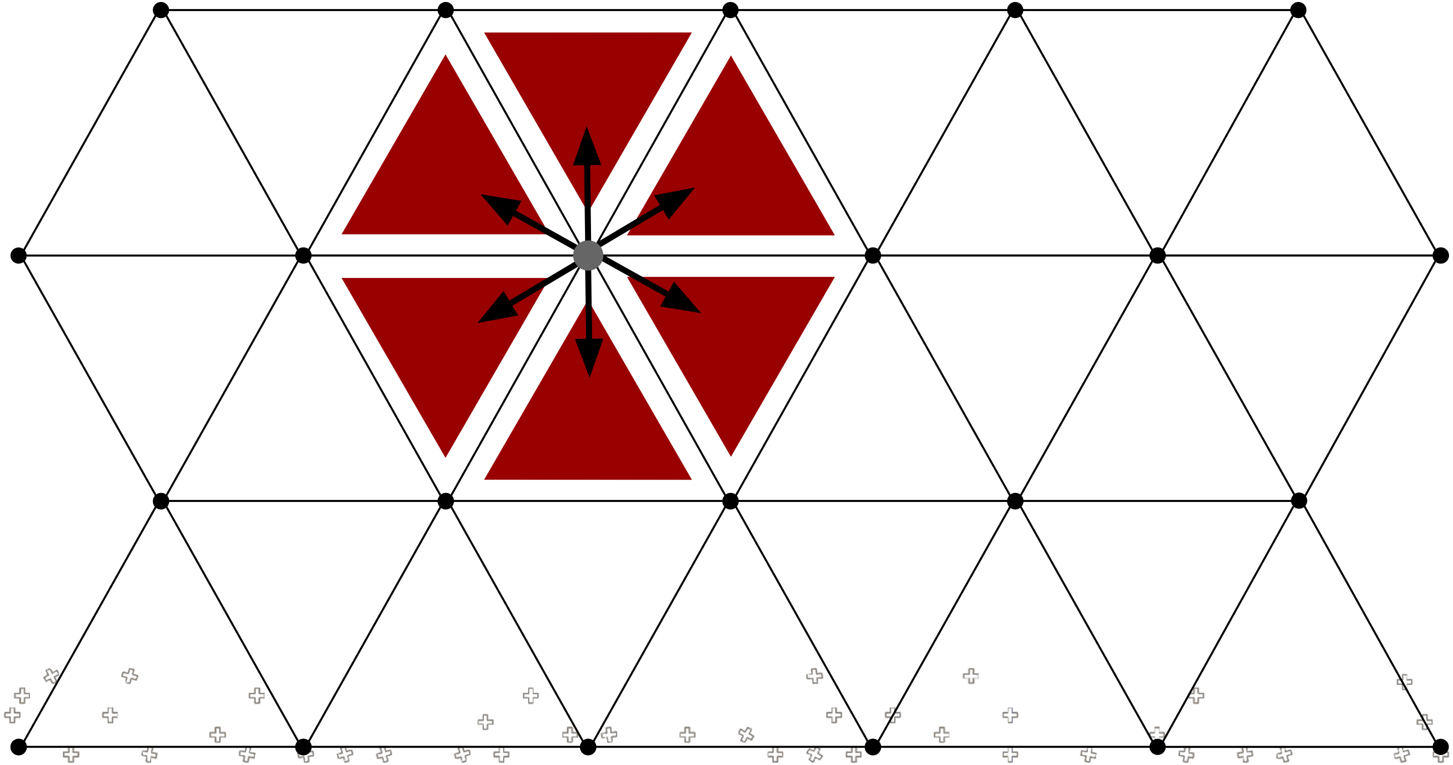
Mesh: Neighbors



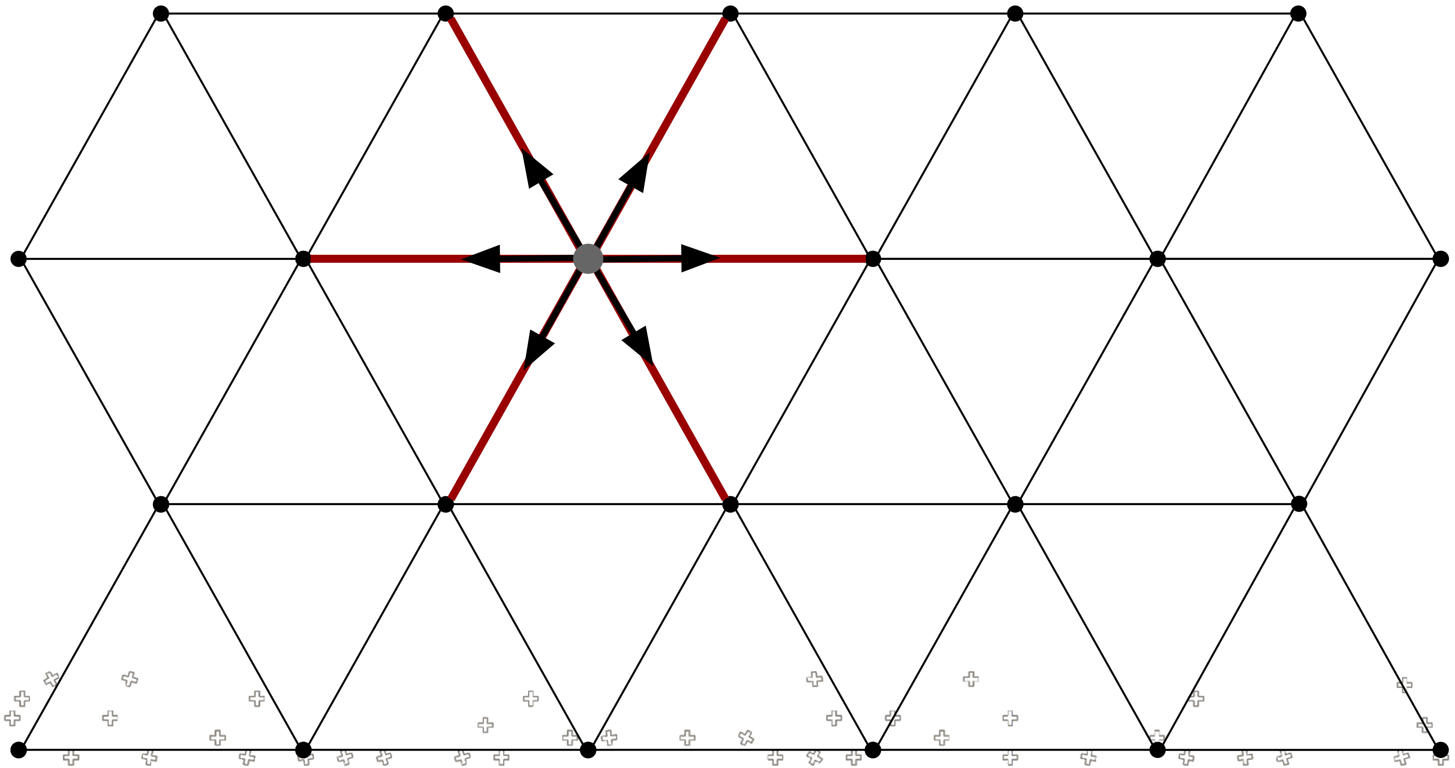
Neighbors: Vertex



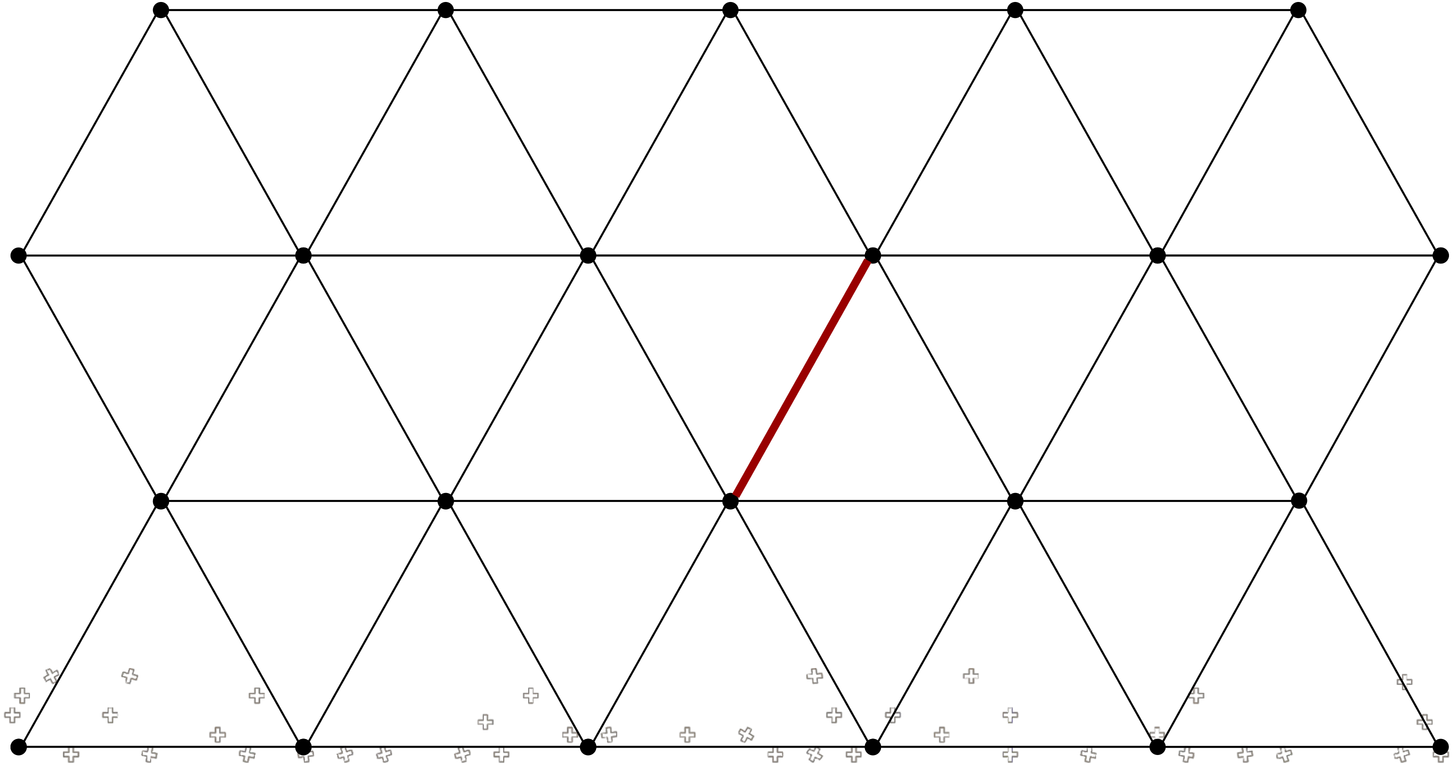
Neighbors: Vertex -> Cell



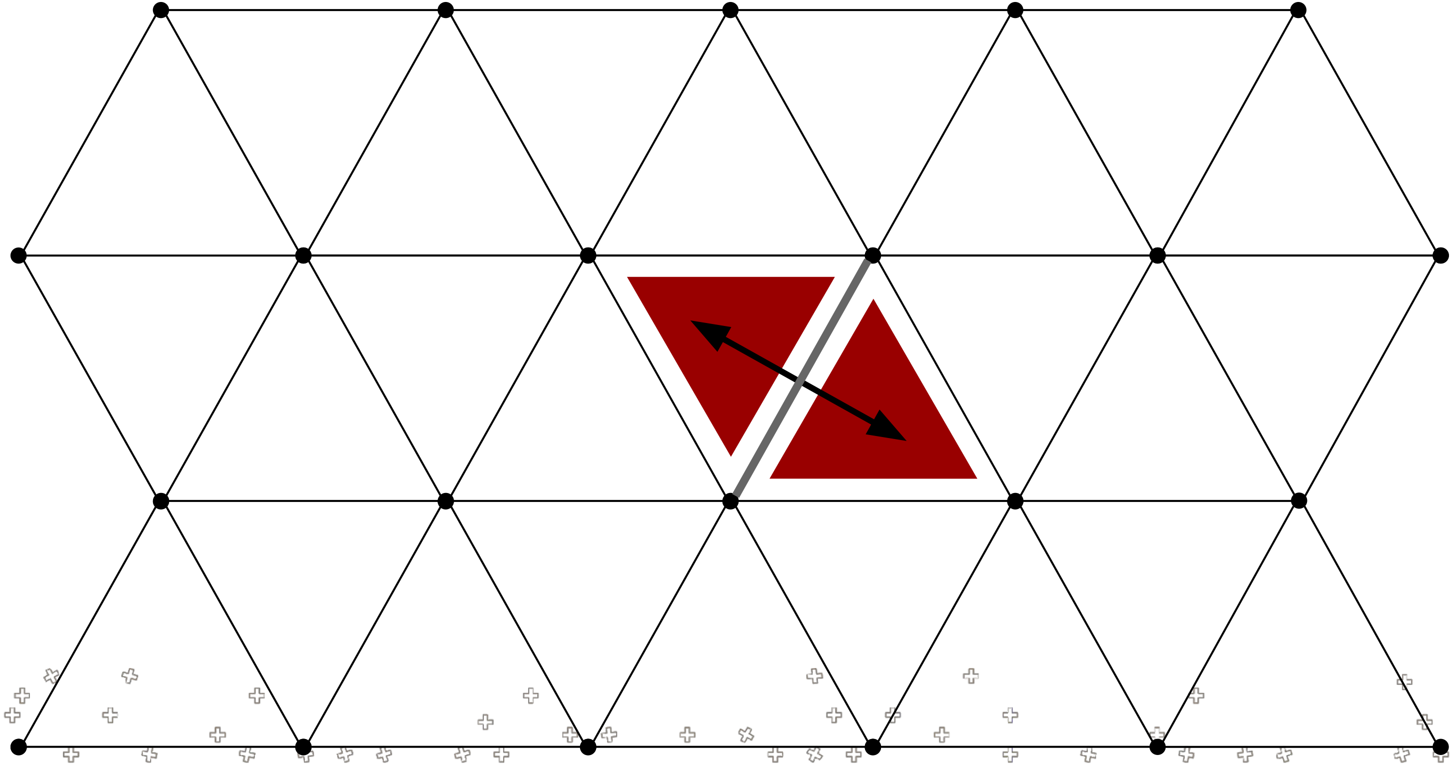
Neighbors: Vertex -> Edge



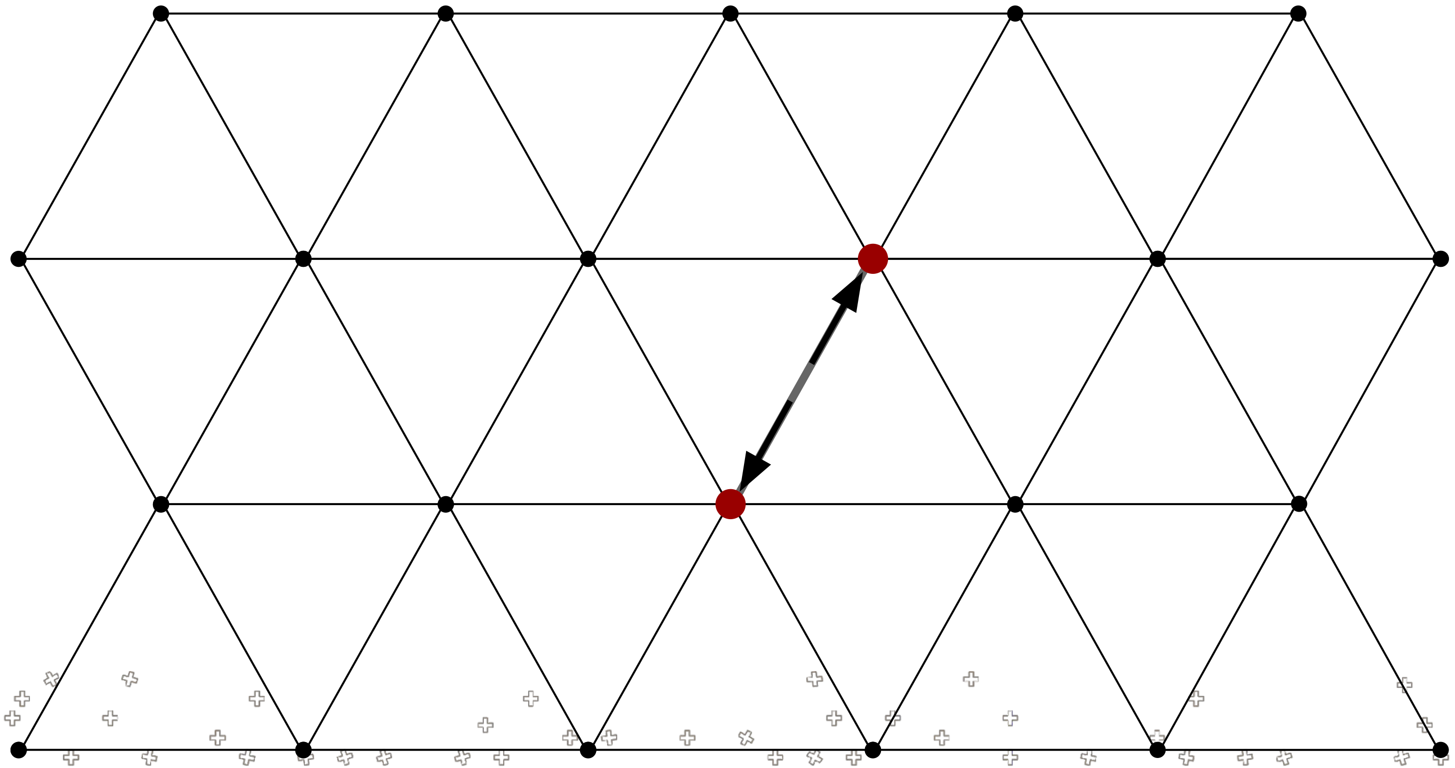
Neighbors: Edge



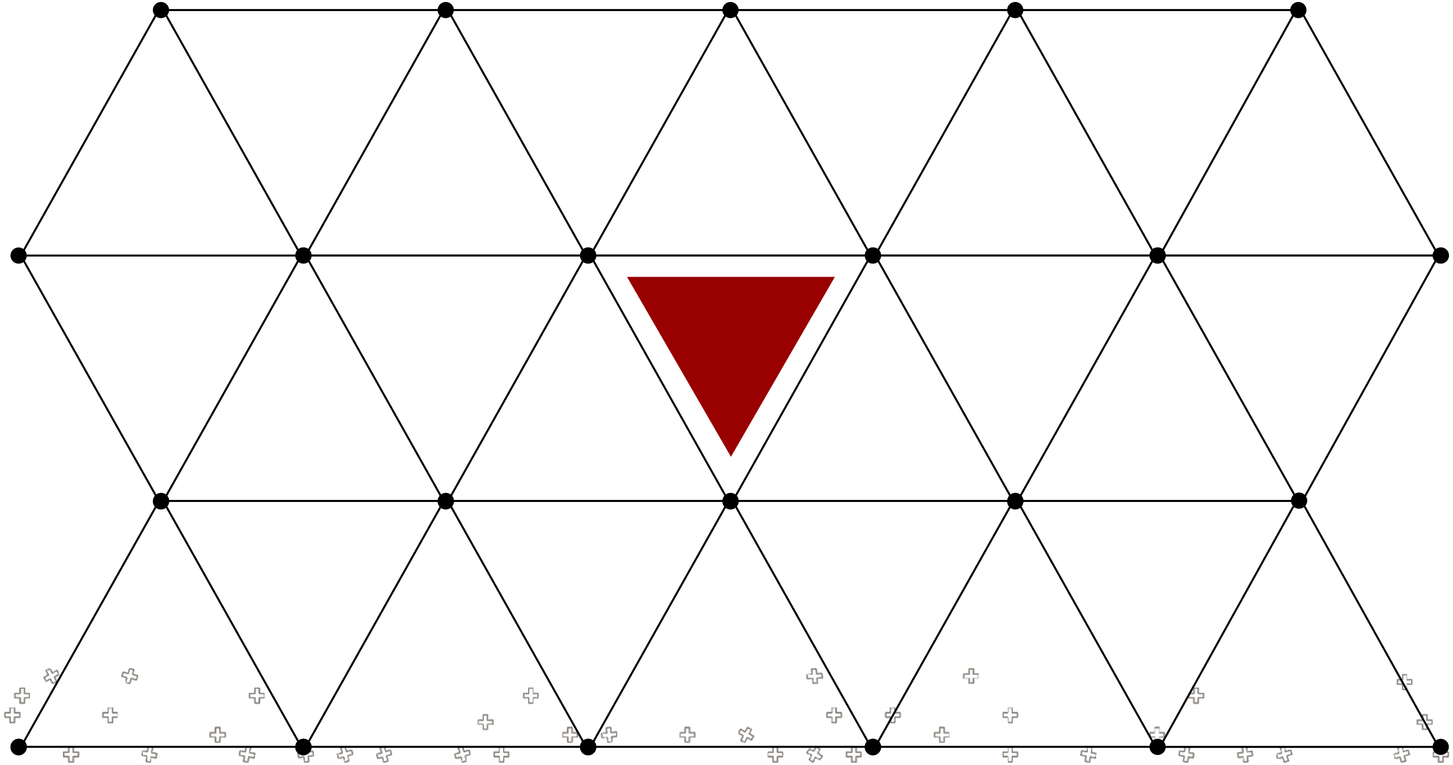
Neighbors: Edge -> Cell



Neighbors: Edge -> Vertex

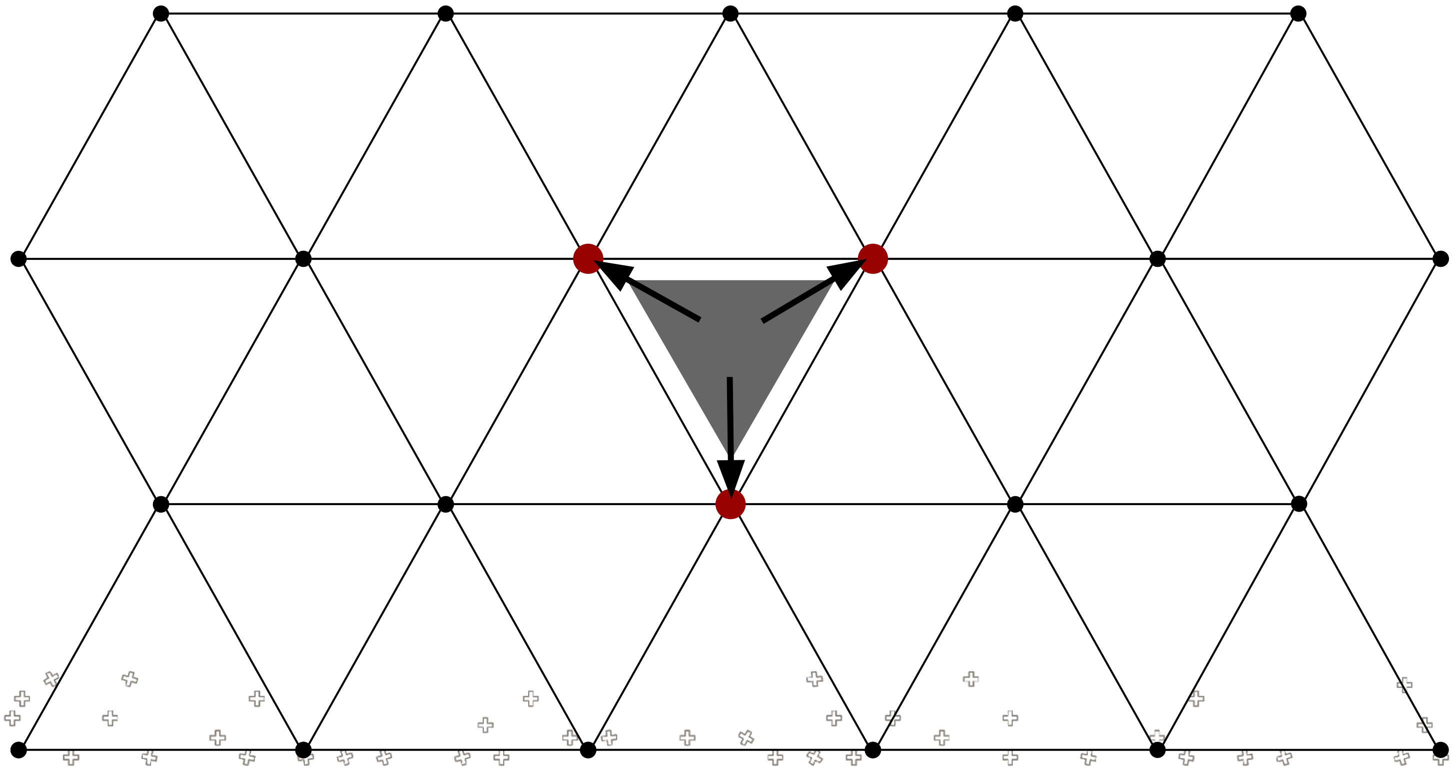


Neighbors: Cell

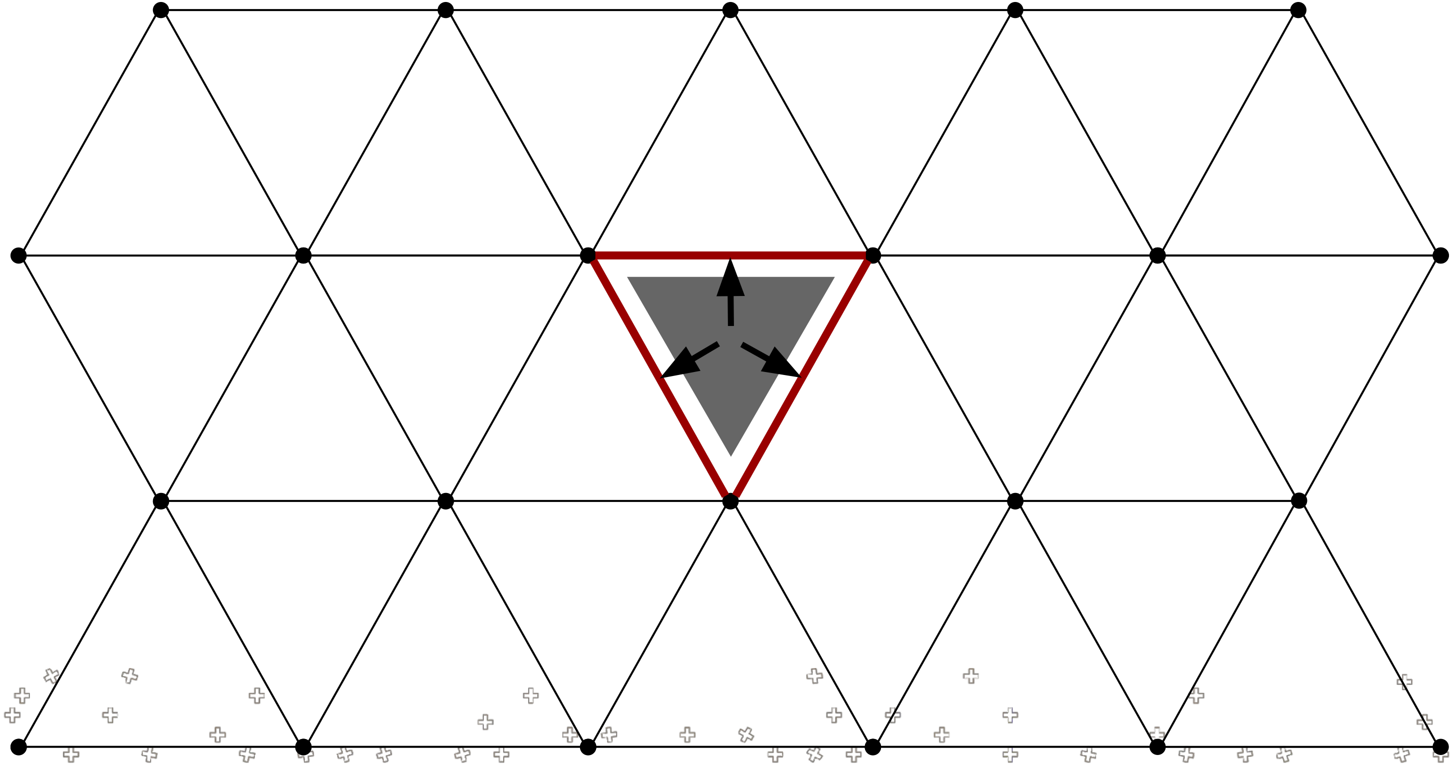


MeteoSwiss

Neighbors: Cell -> Vertex



Neighbors: Cell -> Edge



Reductions - Neighborhood

- For now, there are the following six neighborhoods
  - Vertex → Cell
  - Vertex → Edge
  - Edge → Cell
  - Edge → Vertex
  - Cell → Vertex
  - Cell → Edge
- There are more general neighborhoods (later)
- The **neighborhood** is the first argument to the dusk reduce primitive

```
@stencil
def reduce(lhs: Field[Edge], rhs: Field[Cell]):
    with levels_downward:
        lhs = reduce_over(Cell > Edge, rhs, sum, init=0.0)
```

Reductions

```
@stencil
def reduce(lhs: Field[Edge], a: Field[Cell], b: Field[Cell]):
    with levels_downward:
        lhs = reduce_over(Edge > Cell, a+b, sum, init=0.0)
```

Neighborhood to iterate over

$$\mathrm{lhs}(e) = \sum_{c=1}^{2} a(c) + b(c)$$

MeteoSwiss

Reductions

```
@stencil
def reduce(lhs: Field[Edge], a: Field[Cell], b: Field[Cell]):
    with levels_downward:
        lhs = reduce_over(Edge > Cell, a+b, sum, init=0.0)
```

Operands - what to do on each (edge) neighbor

$$\mathrm{lhs}(e) = \sum_{c=1}^{2} a(c) + b(c)$$

MeteoSwiss

Reductions

```
@stencil
def reduce(lhs: Field[Edge], a: Field[Cell], b: Field[Cell]):
    with levels_downward:
        lhs = reduce_over(Edge > Cell, a+b, sum, init=0.0)
```

Operator - how to "combine" the values computed at the (cell) neighbors (in this case sum up)

$$\text{lhs}(e) = \sum_{c=1}^{2} a(c) + b(c)$$

Reductions

```
@stencil
def reduce(lhs: Field[Edge], a: Field[Cell], b: Field[Cell]):
    with levels_downward:
        lhs = reduce_over(Edge > Cell, a+b, sum, init=0.0)
```

Initial Value - Value to start the summation with

$$\text{lhs}(e) = \sum_{c=1}^{2} a(c) + b(c)$$

Reductions

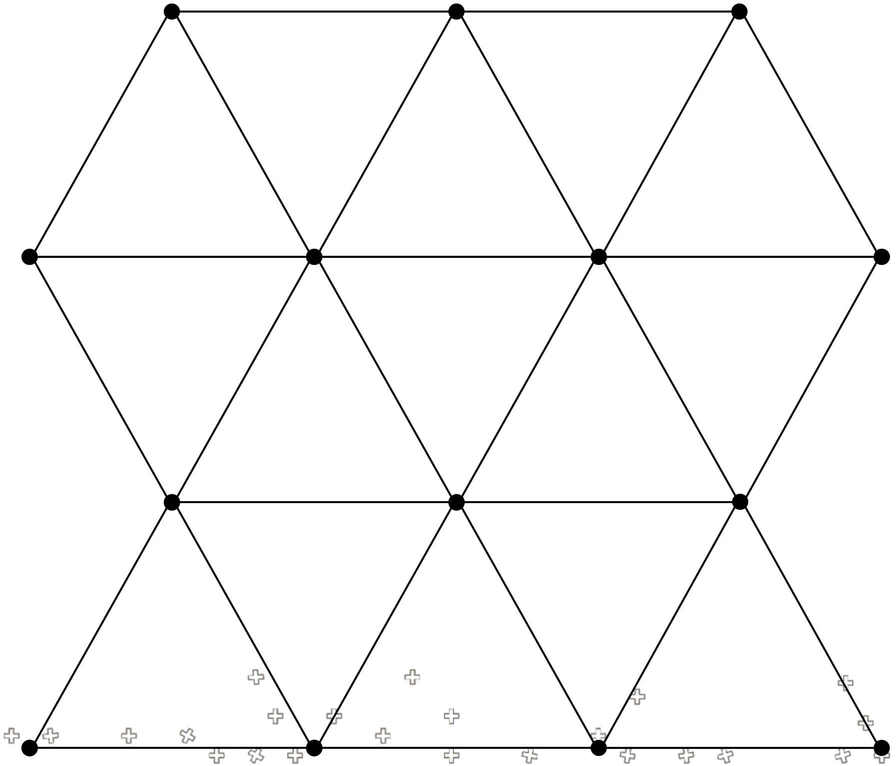
```
@stencil
def reduce(lhs: Field[Edge], a: Field[Cell], b: Field[Cell]):
    with levels_downward:
        lhs = sum_over(Edge > Cell, a+b)
```

shorthand for `reduce_over(Edge > Cell, …, sum, init = 0)`

$$\mathrm{lhs}(e) = \sum_{c=1}^{2} a(c) + b(c)$$

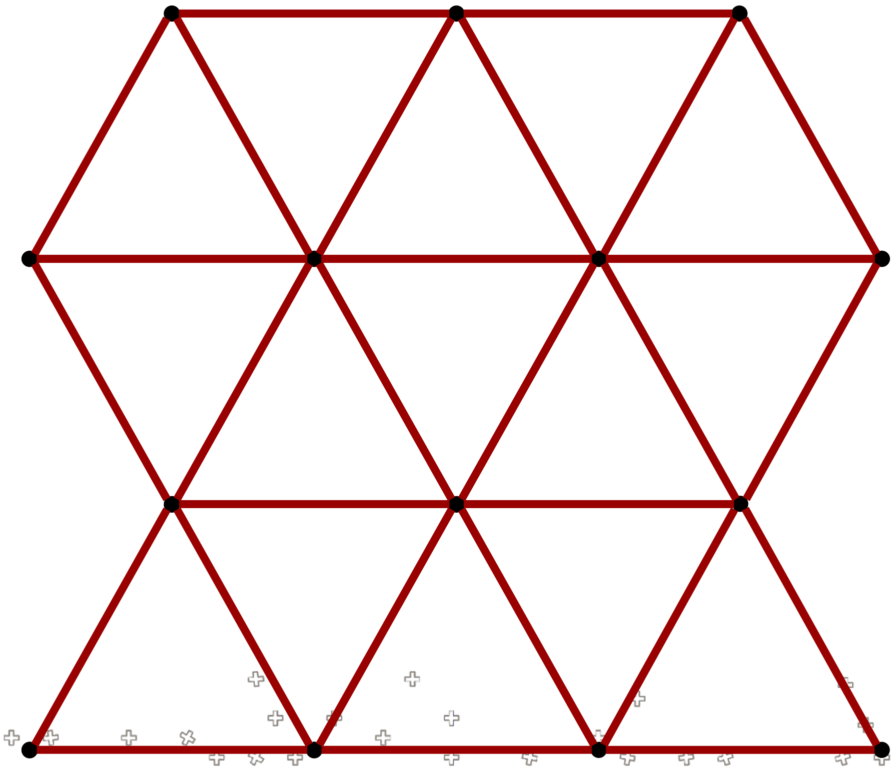
Reduction - Animated Example

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
   with levels_downward:

       lhs = sum_over(Edge > Cell, a+b)
```



Reduction - Animated Example

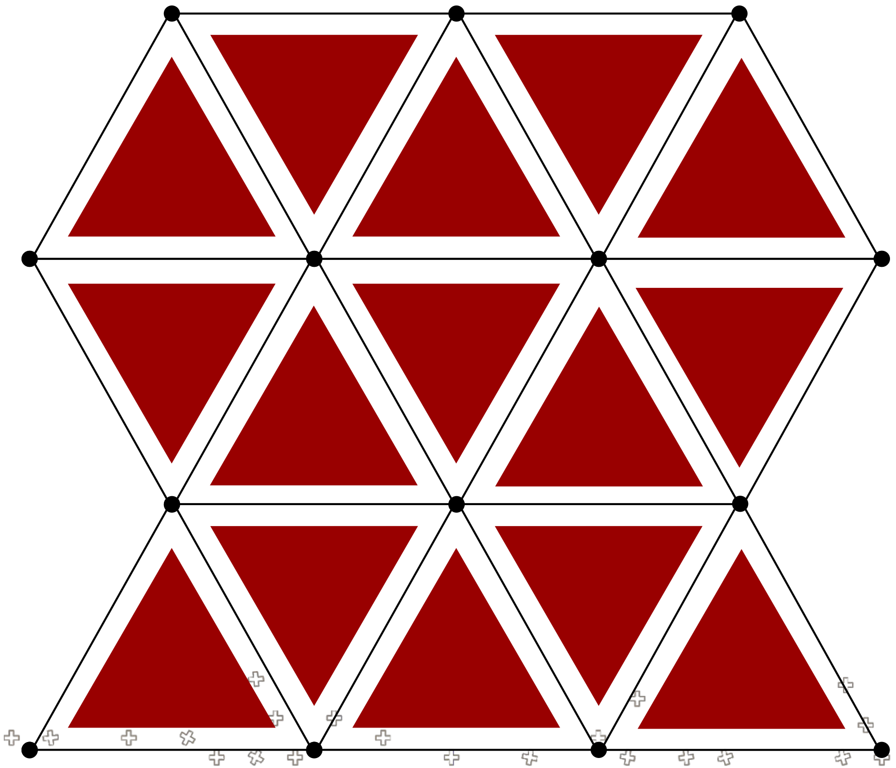
```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
  with levels_downward:

     lhs = sum_over(Edge > Cell, a+b)
```



MeteoSwiss

Reduction - Animated Example

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
  with levels_downward:
      lhs = sum_over(Edge > Cell, a+b)
```

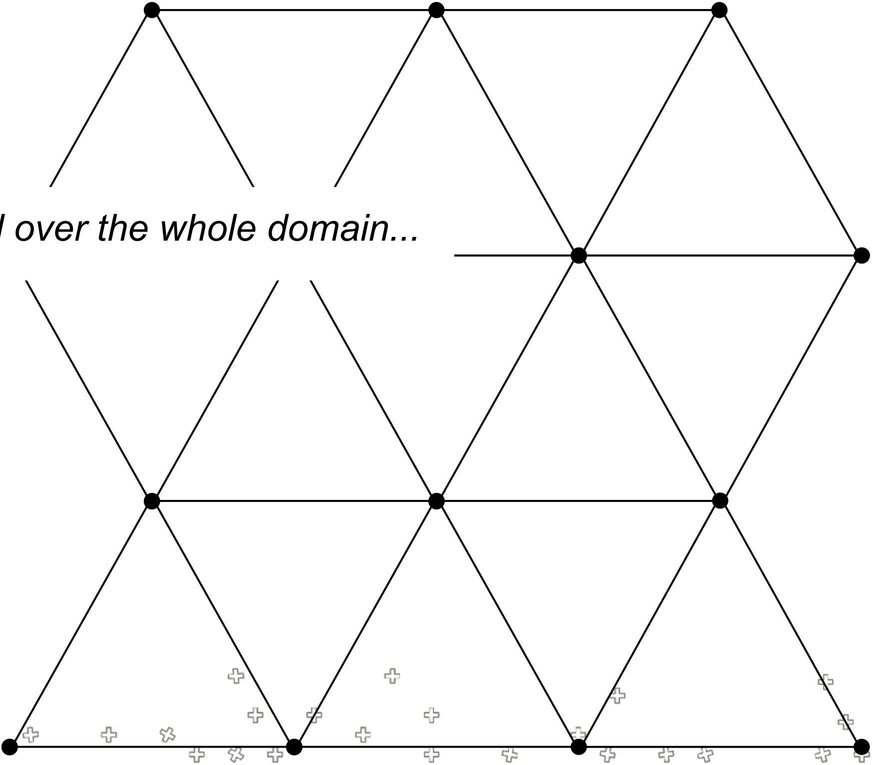


MeteoSwiss

Reduction - Animated Example

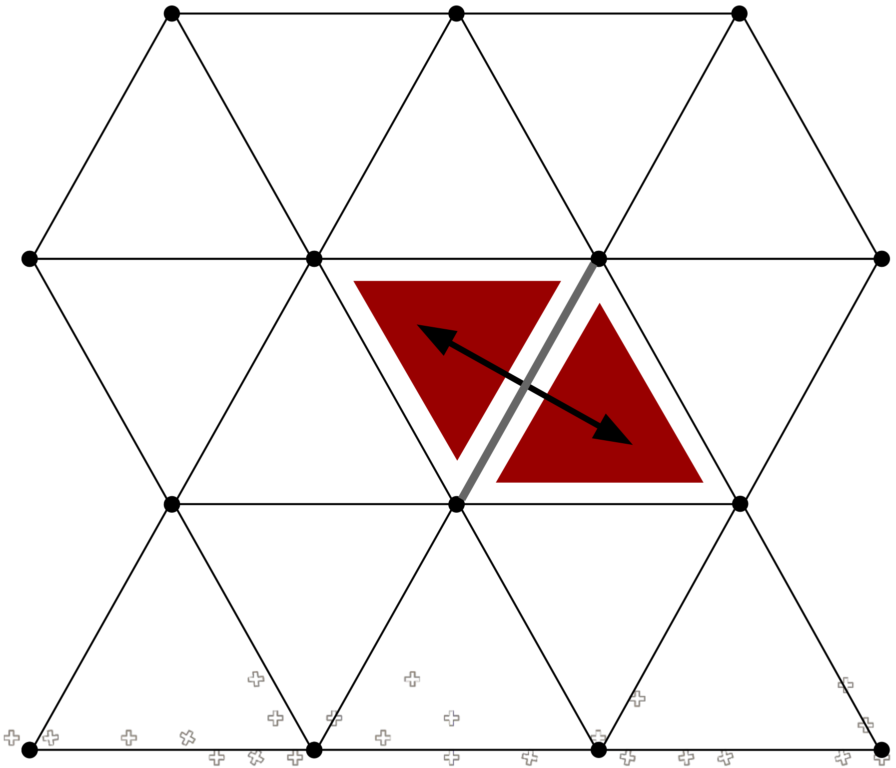
```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
    with levels_downward:
        lhs = sum_over(Edge > Cell, a+b)
```

Run the stencil over the whole domain...



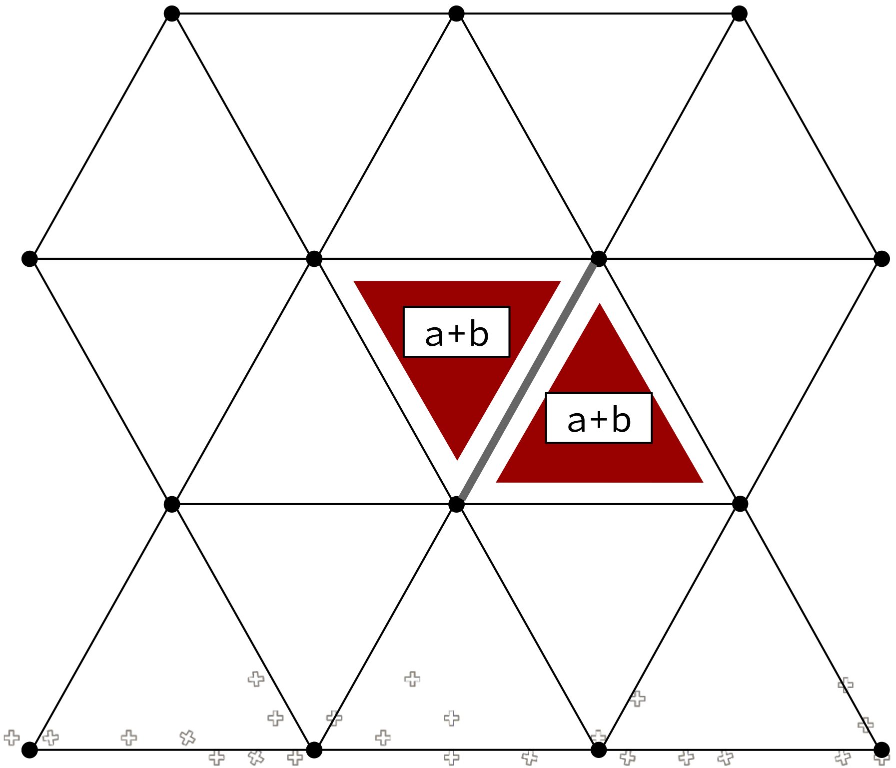
Reduction - Animated Example

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
  with levels_downward:
    lhs = sum_over(Edge > Cell, a+b)
```



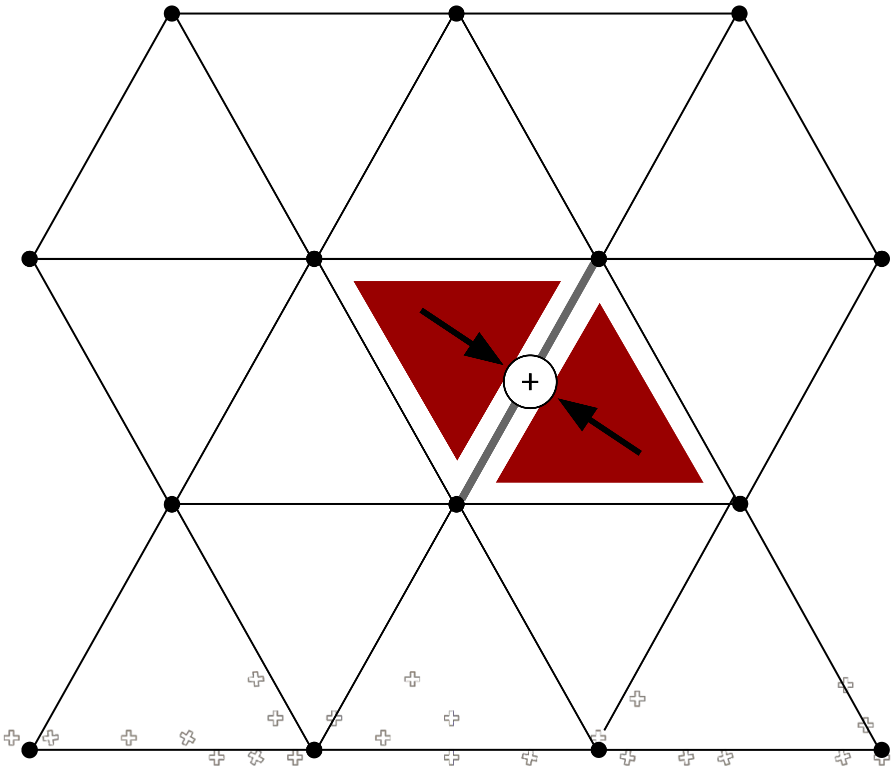
Reduction - Animated Example

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
    with levels_downward:
        lhs = sum_over(Edge > Cell, a+b)
```



Reduction - Animated Example

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
   with levels_downward:
       lhs = sum_over(Edge > Cell, a+b)
```



Reduction - Emitted Pseudo Code

```
@stencil
def reduce(lhs: Field[Edge],
           a: Field[Cell],
           b: Field[Cell]):
  with levels_downward:

    lhs = sum_over(Edge > Cell, a+b)
```

dawn

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
  for (cIdx : mesh.nbh_cells(eIdx))
    lhs(eIdx) += a(cidx) + b(cIdx)
```

Reduction - Emitted Pseudo Code

```
@stencil
def reduce(lhs: Field[Edge,K],
           a: Field[Cell,K],
           b: Field[Cell,K]):
   with levels_downward:

       lhs = sum_over(Edge > Cell, a+b)
```

dawn

```
parfor (k = 0; k < kmax; k++)
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      for (cIdx : mesh.nbh_cells(eIdx))
            lhs(eIdx,k) += a(cidx,k) + b(cIdx,k)
```

Reductions - Using Weights

- Sometimes it is useful to scale each operand in a reduction by some weight
- The dusk reduction concept supports this idea using the optional keyword argument `weights`
- The following two snippets are equivalent

```
@stencil
def reduce(lhs: Field[Edge], rhs: Field[Cell],
           w: Field[Edge]):
  with levels_downward:
      lhs = sum_over(Edge > Cell, rhs) / w
```

↔

```
@stencil
def reduce(lhs: Field[Edge],
           rhs: Field[Cell], w: Field[Edge]):
  with levels_downward:
      lhs = sum_over(Edge > Cell, rhs,
                     weights=[1/w, 1/w])
```

- Note that the user is responsible to ensure the weights vector is of the correct length. Here two entries are appropriate since each edge has two cell neighbors
- Here, we didn't gain anything by using weights. Quite the contrary, one might argue that the left hand version is clearer

Reductions - Using Weights

So what are some more realistic / useful use cases for weighted reductions?

- *Directional* gradient along an edge normal

```
@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
```

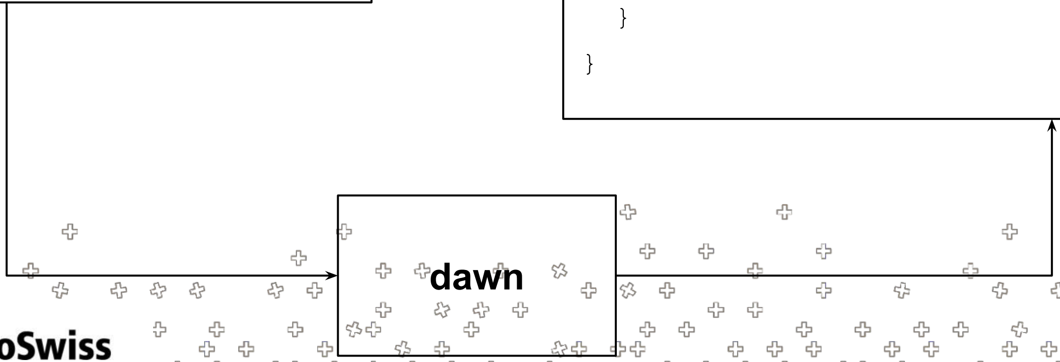
- Interpolation from two locations to one with pre-computed interpolation weights

```
@stencil
def intp(fe: Field[Edge], alpha: Field[Edge], fc: Field[Cell]):
    with levels_downward:
        fe = sum_over(Edge > Cell, fc, weights=[1-alpha,alpha])
```

- Becomes more useful with later advanced concepts

Weighted Reduction - Emitted Pseudo Code

```
def intp(fe: Field[Edge,K],
         alpha: Field[Edge,K],
         fc: Field[Cell,K]):
   with levels_downward:
      fe = sum_over(Edge > Cell, fc,
                    weights=[1-alpha,alpha])
```




```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      weights = {1-alpha(eIdx, k),
                 alpha(eIdx, k)}
      linear_idx = 0
      for (cIdx : mesh.nbh_cells(eIdx)) {
            fe(eIdx,k) += fc(cidx,k)*weights[linear_idx]
            linear_idx++
      }
  }
}
```

Reductions - Short Hands

We have already seen one shorthand notation:

```
@stencil
def reduce(out: Field[Vertex], in: Field[Edge])
    with levels_downward:
        out = reduce_over(Vertex > Edge, in, sum, init=0)
        out = sum_over(Vertex > Edge, in)
```

There are two others to find the minimum and maximum

```
@stencil
def reduce(out: Field[Vertex], in: Field[Edge])
    with levels_downward:
        out = min_over(Vertex > Edge, in)
        out = max_over(Vertex > Edge, in)
```

Conditionals & Control Flow

Often one wants to execute certain computations only if some conditions hold. Some simple examples:

- boundary conditions
- only run a damping method in parts of the field which are oscillatory
- only perform computations in parts of a field which are given by a pre-computed mask

Just as in about any other programming language, this mechanism is realized using an if-then-else concept:

```
@stencil
def control_flow(f: Field[Edge]):
   with levels_downward:
      if f < 10:
            f = f + 10
      else:
            f = f + 5
```

→ **dawn** →

```
for (eIdx = 0; cIdx < mesh.num_edges(); eIdx++)
    if(f[eIdx] < 10 ) {
      f[eIdx] += 10
    } else {
      f[eIdx] += 5
    }
```

Conditionals & Control Flow

Only caveat

- as stated dusk & dawn do not support boolean fields yet
- masks need to be emulated using floats
- probably the safest option is to use 0. for false and 1. for true

```
@stencil
def control_flow(f: Field[Edge], mask: Field[Edge]):
    with levels_downward:
       if (mask == 1):
            f = f + 10
       else:
            f = f + 5
```

Wrap Up / Repetition

What can we do in dawn so far?

We can conveniently do arithmetic on fields

```
@stencil
def math(a: Field[Edge, K], b: Field[Edge, K], c: Field[Edge, K]):
    with levels_downward:
        a = b / c + 5
```

Wrap Up / Repetition

What can we do in dawn so far?

We can introduce control flow

```
@stencil
def bnd_cond(vx: Field[Edge, K], vy: Field[Edge, K], boundary_edges: Field[Edge, K]):
    with levels_downward:
        if (boundary_edges == 1.):
                vx = 0
                vy = 0
        else:
                #evolve vx, vy
```

Wrap Up / Repetition

What can we do in dawn so far?

We can reduce from one location type to another

```
@stencil
def average(fc_avg: Field[Cell, K], fe: Field[Edge, K]):
    with levels_downward:
        fc_avg = sum_over(Cell > Edge, fe) / 3 #3 edges per cell
```

Wrap Up / Repetition

What can we do in dawn so far?

We can weight these reductions

```
@stencil
def average(fc_avg: Field[Cell, K], fe: Field[Edge, K]):
    with levels_downward:
        fc_avg = sum_over(Cell > Edge, fe, weights=[1/3, 1/3, 1/3]) #3 edges per cell
```

MeteoSwiss

Wrap Up / Repetition

- dawn makes sure that the code can be run in parallel safely
  - code that can not be run safely in parallel is emitted as sequential code[1]
- user needs to make sure that code is type consistent
  - respect dimensionality / location
  - dawn rejects inconsistent code

1) currently some edge cases are still rejected instead of emitted sequentially

Wrap Up / Repetition

- The combination of these concepts is already quite powerful
- Powerful enough in fact to compute various quantities in (vector) analysis: gradient, divergence, …

  →see exercise

- In the next session more advanced dusk & dawn concepts will be presented

Q&A

Questions?