New Field Abstractions

Jeffrey D. Oldham and Stephen Smith

2001 Jul 23

Contents

Jim Crotinger, Scott Haney, Jeffrey D. Oldham, and Stephen Smith participated in this design.

1 Fundamental Field Concepts

- A field is an array in space. That is, given an index into a field, one can ask for its position in, e.g., 3D, space.
- A field cell is a field's fundamental building block. All field concepts are cell-based.
- The lower, left corner of a cell in d-D space is its origin 0^d. The upper right corner has coordinates 1^d.
- A field can store at most one value at each position within each cell.

2 Centering Specifications

A field's centering is specified by an object with the following fields:

centering type: an enumeration for vertex, edge, face, or cell centering type

discontinuous: a boolean indicating whether, for a value located on a cell boundary, each of the neighboring cells has its own value (*discontinuous*) xor one value for all neighboring cells is provided (*continuous*)

list of values: Each list element is a pair of an orientation and a position. The position, a d-D real vector, specifies the value's position with respect to the cell's coordinate system, which is either $[0,1)^d$ or $[0,1]^d$ depending on whether values are continuous or discontinuous, respectively. The orientation in Z_2^d, indicates which zeroes (or ones if discontinuous) in the position must be zero (or one) because of the centering type. For example, a continuous face centering for an x-face must have a 0 in the x-component. Other coordinates can be zero but need not be.

For a d-D cell, a face is a $(d-1)$-D object. An edge centering is always a 1-D object. For two dimensions, edge and face centerings are the same although it is sometimes useful to distinguish them when writing programs that work for various dimensions.

Adjacent cells can share values. For example, a vertex-centered value in a 3-D field is shared by eight cells. A program might require that each cell maintain its own value at the point. To do so, specify discontinuous values. To be a valid discontinuous centering, values on cell boundaries must be arranged so that every adjacent cell also has a value at that same position in space. For example, any discontinuous edge value in three-dimensional space must have related values in the three adjacent cells.[1]

Each value in the list should be specified exactly once. For example, the canonical continuous vertex-centered value is 0^d. This cell specifies a value at its origin. Neighboring cells specify values at their origins. Collectively, all the field's vertex values are specified. For continuous values, positions should be in the range $[0,1)^d$. For discontinuous values, positions should be in the range $[0,1]^d$. To implement the field, each specified position corresponds to a subfield.

Orientations are used when creating storage for field values. When creating storage, the number of values does not necessarily match the number of cells. For example, to create a field with n^2 cells requires $(n+1)^2$ vertices. Although orientations might appear to be redundant since nonzero values indicate an orientation, an x-face centered value may have position $(0,0,0.5)$, which does not indicate whether it is an x- or y-face.

[1] We need to provide a justification for this restriction.

2.1 Examples

2.1.1 Orientation Examples

For three-dimensions, orientations for

vertex type: 000 since vertex positions are completely determined

x**-edge:** 100 since the x-coordinate varies along the edge

x**-face:** 011 since the x-coordinate is fixed at zero (or one) but the other values can vary

cell type: 111 since values can be placed at any location.

2.1.2 Centering Examples

Continuous Vertex Centering Values occur at every vertex in the field, but each cell is responsible only for the vertex located at its origin. Thus, the centering need only declare one orientation-position pair:

Vertex false /* continuous */ {((0, 0, 0), (0.0, 0.0, 0.0))}

Discontinuous Vertex Centering Each cell maintains its own set of 2^d vertex values.

Vertex true /* discontinuous */ {((0, 0, 0), (0.0, 0.0, 0.0)),
((0, 0, 0), (1.0, 0.0, 0.0)),
((0, 0, 0), (0.0, 1.0, 0.0)),
((0, 0, 0), (0.0, 0.0, 1.0)),
((0, 0, 0), (1.0, 1.0, 0.0)),
((0, 0, 0), (1.0, 0.0, 1.0)),
((0, 0, 0), (0.0, 1.0, 1.0)),
((0, 0, 0), (1.0, 1.0, 1.0))}.

Continuous Face Centering Since the faces are shared, each cell is responsible for only the faces intersecting the origin. Here we specify one value on each face.

Face false /* continuous */ {((0, 1, 1), (0.0, 0.5, 0.5)),
((1, 0, 1), (0.5, 0.0, 0.5)),
((1, 1, 0), (0.5, 0.5, 0.0))}.

2.2 Centering Design Criteria

- To facilitate writing programs that work for various dimensions, the centering type should be specified and stored. For example, in two dimensions, edge and face centerings are the same, but a program may require a face centering. Thus, face, not edge, should be stored.

3 Pooma Field Computation Notation

When writing Pooma field computations, notation for accessing field values is needed. Since Pooma is cell-centric, all value locations are relative to a particular cell and its coordinate system. Most computations use one or more cells centered around one particular cell. Below, we assume its indices are specified by a `Loc` called `loc`, and we refer to all cell locations relative to it. Given `loc`, any field value can be specified by a `FieldOffset`, which is a cell offset (relative to `loc`) and the number of a value within the cell.

cell offset: This Z^d value specifies the value's cell relative to `loc`, which is conceptually at the origin 0^d. For example, in a three-dimensional field, the cells to the left and right are $(-1, 0, 0)$ and $(1, 0, 0)$, respectively. In Pooma, we will represent a cell offset using a `Loc<dim>`.

value number: All the values in a cell specified by a centering are numbered according to the order of declaration in the centering. For example, the zeroth value specified in the centering has number 0. The next value (if present) has number 1.

Given a field `f`, a `Loc loc`, and a field offset `(offset,num)`, a field value can be obtained. Since each value specified by the field's centering is stored in a separate subfield, the notation `f[num](loc + offset)` yields the value. Alternative notation permits operating on a field using a `FieldOffset`: `f(FieldOffset, Loc<dim>)` yields a value at the same position. `f(FieldOffset)` yields a comparable data-parallel statement for all cells.

Some output centerings have vertices shared among adjacent cells. Pooma computations may use any input field values but should assign only to values specified in the output centering. Since the output centering includes only values for which the cell is responsible, assigning only to these values prevents assignment from multiple cells.

3.1 Different Pooma Computations

Pooma supports three different computation paradigms: scalar, data parallel, and stencil-based. To illustrate these three styles, we use a scatter computation copying a cell-centered value to each output value in a cell. For this example, the output centering does not matter.

3.1.1 Scalar Computation

A scalar computation explicitly assumes all output field values.

```
???some loop over loc???
  for (int i = outputField.centering().size() - 1; i >= 0; --i)
    out[i](loc) = in[0](loc);
```

FINISH: What is the right `loc` treatment?

3.1.2 Data Parallel Computation

For data parallel computations, the same computation is applied to every field location. Thus, `loc` is omitted.

```
for (int i = outputField.centering().size() - 1; i >= 0; --i)
  out[i] = in[0];
```

3.1.3 Stencil Computation

Even though an output field cell may have multiple values, each stencil computation yields one output value. The stencil operator creates one stencil computation for each output value. Thus, the stencil computation can assume it is given `loc` and a destination field value number. The computation

```
return in[0](loc);
```

ignores the output value number since all values receive the same (and only) input value.

4 Easing Pooma Computation Implementations

Although the `FieldOffset` notation is sufficient to write any computation, we also provide sets of values and reduction operations to ease writing Pooma field computations in a dimension-independent way.

The most common computations used in the Caramana hydrodynamics program and in the examples we created involve computations using the input values closest to the computed output value. Here we assume only one input field. Also, we use the logical coordinate space where a cell's origin has coordinate 0^d and its most extreme corner has coordinate 1^d, regardless of its physical size.

To compute an output value's nearest neighbors, align the logical coordinate spaces of the input and output field's cells. Starting at the output value, expand a ball using the Manhattan norm ℓ_1 until it touches one or more input field values. The ball's radius may be any nonnegative number. We call this smallest ball the *first shell*. We make these available in a `FieldOffsetList`.

The most trivial example is computing one cell-centered value from an input field with a cell-centered value at the same location in the logical coordinate space. Expanding the ball immediately touches the input field value using a radius of zero so it is the only `FieldOffset` in the list.

Computing a continuous vertex value by summing all discontinuous vertex values demonstrates that more than one input field can be involved. More than one of the input field's discontinuous vertex values may be located at the same logical coordinate location, but each is associated with a distinct cell. Expanding a ball starting at the output field value immediately touches all the discontinuous values associated with the same location. All these are included in a `FieldOffsetList`.

Forming a cell-centered value by summing all discontinuous vertex values within a cell illustrates that `FieldOffsetList`s can be restricted to those within a cell. An expanding ball starting from a cell's center first touches input field values at the cell's corners but it also touches the discontinuous vertex values at the adjacent cells since they are at the same logical coordinate locations. Specifying including values only from the input field cell corresponding to the output field cell in the `FieldOffsetList` excludes them.

Future work may permit specifying values on other shells or the union of these values.

4.1 `FieldOffsetList` Operations

A `FieldOffsetList` contains a sequence of `FieldOffset`s similar to a fixed-length `vector<FieldOffset>`. Among the supported operations are

`.size()`: returning the nonnegative number of `FieldOffset`s.

`operator[](n)`: returning a constant reference to the `FieldOffset` located at the specified zero-based location.

In fact, we will implement the class as `std::vector<FieldOffset>`.

4.2 Computing the Values in the First Shell

We explain how to compute the input values in the first shell given the input and output centerings and a specified output value. We wish to determine all input field value locations v_i which minimize $d_1(v_i, v_o)$, where d_1 is the Manhattan distance and v_o is the location of the output field value. More precisely, we wish to minimize the absolute value of $(v_{ic} - v_o) \pmod{1^d}$, where v_{ic} varies over all input centering locations.

Incorporating the modulus reflects the repetition of input values throughout the field. Since each cell has at least one input field value, this set is non-empty and also the largest distance to consider is $d * 0.5$.[2]

To compute the absolute value over all input centering locations, we compute an absolute value for each location, remembering the associated locations. Since we are using the Manhattan norm, each dimension can be treated independently. If $|v_{ic} - v_o|$ restricted to the dimension is greater than 0.5, add one or subtract one to yield an absolute value less than 0.5. If the value equals 0.5, add one or subtract one to determine another location with the same distance. As the minimum computes through all input centering locations, only the minimum distance computed so far and the associated values need be stored.

4.3 Computations Involving Reductions

Many computations involve reducing the values specified in `FieldOffsetList`s. For example, an output value can be the average of the input field values.

[2]Proof: A cell's width is at most $d * 1$. Since cell values are repeated every distance 1 in every dimension, the largest distance between an output value location and one of the repeated input locations is actually 0.5 in any dimension.

Thus, we will provide a canonical set of reduction functions, each taking a field, a `FieldOffsetList`, and a cell location `loc`. The reduction functions will include `sum`, `average`, `min`, and `max`. Each reduction function iterates through the list of `FieldOffset`s, using the field values located at the sums of the `FieldOffset`s and `loc`. These are dimension-independent computations, which may use different numbers of values for different dimensions. Thus, `average` is needed in addition to codesum.

4.4 Computing `FieldOffsetLists`

Instead of computing `FieldOffsetList`s when needed, we precompute commonly used lists. Given an input centering and an output centering, the `findFieldOffsetList` function returns a `std::vector` of `FieldOffsetList`s. Each `vector` entry corresponds to one output value in the output centering.

Eventually, the function will use a `std::map` from the pair of an input centering and an output centering to a `std::vector<FieldOffsetList`. Each centering will be given an ID. To form the ID, the `Centering` class will be a reference counter pointer to a `CenteringData` object. The ID will the pointer's value.

4.5 Summing Discontinuous Vertex Values to a Cell Center

input centering: discontinuous vertex values. For three dimensions, the centering is

Vertex discontinuous {((0, 0, 0), (0.0, 0.0, 0.0)),
((0, 0, 0), (1.0, 0.0, 0.0)),
((0, 0, 0), (0.0, 1.0, 0.0)),
((0, 0, 0), (0.0, 0.0, 1.0)),
((0, 0, 0), (1.0, 1.0, 0.0)),
((0, 0, 0), (1.0, 0.0, 1.0)),
((0, 0, 0), (0.0, 1.0, 1.0)),
((0, 0, 0), (1.0, 1.0, 1.0))}.

output centering: one cell centered point:

Cell continuous {((1, 1, 1), (0.5, 0.5, 0.5))}

computation: Sum all vertex values.

Syntax showing a loop:

```
double result = 0.0;
for (FieldOffsetIterator<dim> iter = CellFromDiscontinuousVert<Dim>();
     iter;
     ++iter)
  result += inputField(*iter, loc);
return result;
```

Shorter syntax:

```
return sum(inputField, CellFromDiscontinuousVert<dim>(), loc);
```

or

```
return sum(inputField, AllCenteringValues<dim>(inputField.centering()), lo
```

4.6 Summing Continuous Vertex Values to a Cell

input centering: continuous vertex values. For three dimensions, the centering is

Vertex continuous {((0, 0, 0), (0.0, 0.0, 0.0))}

output centering: one cell centered point:

Cell continuous {((1, 1, 1), (0.5, 0.5, 0.5))}

computation: Sum all vertex values.

```
return sum(inputField, CellFromVert<dim>(), loc);
```

The iterator `CellFromVert` produces a sequence of field offsets visiting all of a cell's vertices, which are shared with adjacent neighbors.

4.7 Summing Discontinuous Vertex Values to a Continuous Vertex

input centering: discontinuous vertex values. For three dimensions, the centering is

Vertex	discontinuous	{((0, 0, 0), (0.0, 0.0, 0.0)), ((0, 0, 0), (1.0, 0.0, 0.0)), ((0, 0, 0), (0.0, 1.0, 0.0)), ((0, 0, 0), (0.0, 0.0, 1.0)), ((0, 0, 0), (1.0, 1.0, 0.0)), ((0, 0, 0), (1.0, 0.0, 1.0)), ((0, 0, 0), (0.0, 1.0, 1.0)), ((0, 0, 0), (1.0, 1.0, 1.0))}.

output centering: continuous vertex values:

Vertex	continuous	{((0, 0, 0), (0.0, 0.0, 0.0))}

computation: Sum all vertex values conceptually on the same point.

```
return sum(inputField, VertFromDiscontinuousVert<dim>(), loc);
```

The iterator visits all the cells adjacent to the given cell's origin, adding their values. For example, from cell (-1, 0, 0), it obtains the value at (1.0, 0.0, 0.0). From cell (-1, 0, -1), it obtains the value at (1.0, 0.0, 1.0).

4.8 Gathering Face Values with Twice the Granularity to Faces

input centering: continuous face-centered values on a grid with twice the granularity of the output grid. For three dimensions, the centering is

Face continuous {((0, 1, 1), (0.0, 0.5, 0.5)),
((1, 0, 1), (0.5, 0.0, 0.5)),
((1, 1, 0), (0.5, 0.5, 0.0))}.

output centering: continuous face-centered vertex values on a grid. The centering is the same.

computation: One output cell corresponds to eight input cells for three dimensions. Each value on the exterior faces of the eight input cells is added to yield the value on the corresponding input field face. The values on interior faces are ignored.

For one stencil output value, we assume are given a `loc` and an orientation `faceOrientation` for a particular face.

```
return sum(inputField, ManyFacesToOneFace<dim,2>(faceOrientation), loc);
```

4.9 Scattering A Value to A Discontinuous Spoke Centering

input centering: cell-centered value. For two dimensions, the centering is

Cell continuous {((1, 1), (0.5, 0.5))}.

output centering: discontinuous spoke edge-centered values. For two dimensions, the centering is

Edge discontinuous {((1, 0), (0.25, 0.0)),
((1, 0), (0.75, 0.0)),
((1, 0), (0.25, 1.0)),
((1, 0), (0.75, 1.0)),
((0, 1), (0.0, 0.25)),
((0, 1), (0.0, 0.75)),
((0, 1), (1.0, 0.25)),
((0, 1), (1.0, 0.75))}

computation: For stencils, we assume we are given a `loc` and a destination field value number.

```
return inputField[0](loc);
```

4.10 Gathering Continuous Spoke Face-Centered to a Cell Center

input centering: continuous spoke face-centered values. For two dimensions, the centering is

Face	continuous	{((1, 0), (0.25, 0.0)), ((1, 0), (0.75, 0.0)), ((0, 1), (0.0, 0.25)), ((0, 1), (0.0, 0.75))}

output centering: one cell centered point:

Cell	continuous	{((1, 1, 1), (0.5, 0.5, 0.5))}

computation: Sum all values and adjacent values.

```
return sum(inputField, SpokeEdgesToCall<dim>(), loc);
```

4.11 Average Cell Centers

input centering: continuous cell-centered values. For two dimensions, the centering is

Cell continuous {((1, 1), (0.5, 0.5))}

output centering: one cell centered point, i.e., the same centering.

computation: Average a cell's nine neighbors (including itself).

```
return sum(in, CellNeighbors<dim>(-1,1), loc) / pow(3, dim);
```

`CellNeighbors<dim>(-1,1)` iterates through cells in the integer range $[-1, 1]^d$.

4.12 Gradient from Cell Centering to Vertex Centering

input centering: cell-centered value. For three dimensions, the centering is

Cell continuous {((1, 1, 1), (0.5, 0.5, 0.5))}.

output centering: vertex-centered value. For three dimensions, the centering is

Vertex continuous {((0, 0, 0), (0.0, 0.0, 0.0))}

computation: Compute the gradient at the cell center from the neighboring vertices.

```
for (FieldOffsetIterator<Dim> cells = cellOfVertNeighbor<Dim>();
     cells; ++cells)
{
  result += inputField(*cells, loc) * (*cells.vectorSign());
}
return result;
```

4.13 Computation of Edge Normals

input centering: vertex-centered value in two dimensions

Vertex continuous {((0, 0), (0.0, 0.0))}.

output centering: edge values. For two dimensions, the centering is

Edge continuous {((1, 0), (0.5, 0.0)),
((0, 1), (0.0, 0.5))}

computation: First there are two stencil engines, one for each of the output centering points. The stencil constructor computes the vertex neighbors:

```
// Typical input:
// outputCenter = ((0, 1), (0.0, 0.5))
// inputCentering = Vertex

Stencil(Center outputCenter,
        Centering inputCentering)
{
  neighborList_m = nearestNeighbor(outputCenter,
                                   inputCentering);
}

// For this example neighborList_m contains:
//   [0](0,1)
//   [0](0,0)
// (In that order, the y-edge returns to the origin.)
```

The stencil application uses the neighbor list to compute the direction:

```
apply(Field f, Loc<2> loc)
{
  Vector<2, double> ret(0);
  for (vert = neighborList_m.begin(); vert; ++vert)
  {
     ret += f(*vert, loc) * (*vert.sign());
  }
  return rotate(ret);
}
```

4.14 Computation of Corner Normals

input centering: edge values. For two dimensions, the centering is

Edge continuous {((1, 0), (0.5 ,0.0)),
((0, 1), (0.0 ,0.5))}

output centering: corner-cell value. For two dimensions, the centering is

Cell continuous {((1, 1), (0.25, 0.25)),
((1, 1), (0.75, 0.25)),
((1, 1), (0.25, 0.75)),
((1, 1), (0.75, 0.75))}

computation: First there are four stencil engines, one for each of the output centering points. The stencil constructor computes the edge neighbors:

```
// Typical input:
// outputCenter = ((1, 1), (0.25, 0.25))
// inputCentering = Edge

Stencil(Center outputCenter,
        Centering inputCentering)
{
  neighborList_m = nearestNeighbor(outputCenter,
                                   inputCentering);
}

// neighborList_m contains something like:
//    [0](0,0)  - lower x-edge
//    [1](0,0)  - left y-edge
```

The stencil application uses the neighbor list to add the two nearest edge normals together for each corner. Note that we stored the normals in a non-discontinuous field, so we need to multiply by the normal sign to get the outward-facing normal.

```
apply(Field f, Loc<2> loc)
{
  Vector<2, double> ret(0);
  for (vert = neighborList_m.begin(); vert; ++vert)
```

```
  {
     ret += f(*vert, loc) * (*vert.normalSign());
  }
  return ret;
}
```

5 Unfinished Work

1. Update the Pooma computation examples.
2. Write down the Caramana hydrodynamics code with illustrations.