POOMA 2.1

Timothy J. Williams
Advanced Computing Laboratory

ACL Seminar
LANL
September 28, 1999

Outline

- Background
- POOMA 2.1
- Field
  - Example: scalar advection
- Particles
  - Example: particle-in-cell
- 2.1 vs. R1

- Details of POOMA 2.1 features

POOMA

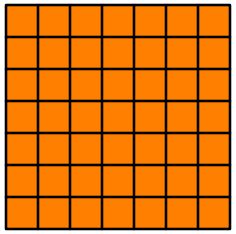
- Parallel Object-Oriented Methods and Applications
  - C++ class library for computational science applications
- POOMA R1
  - Fields, particles, Cartesian meshes, operators, parallel I/O, PAWS
    - (Encapsulated) message-passing
  - Example uses
    - Beneath Tecolote framework in Blanca Project
    - Accelerator physics Grand Challenge
- POOMA 2
  - Redesign, reimplement from scratch
    - SMARTS thread-based parallelism
  - **http://www.acl.lanl.gov/pooma**
    - Tutorials, Presentations (these slides)

Project 2.1

- Recover most POOMA R1 capabilities
- Build on POOMA 2.0.x **`Array`**
  - Map {indices} $\longrightarrow$ value

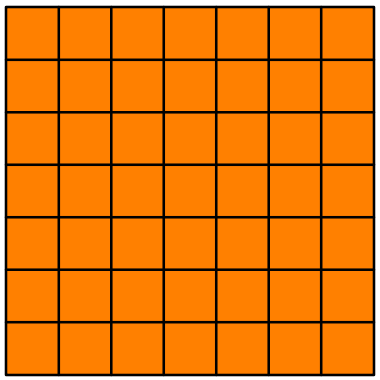
$$(i_1, i_2, \ldots, i_N) \longrightarrow \text{value}$$

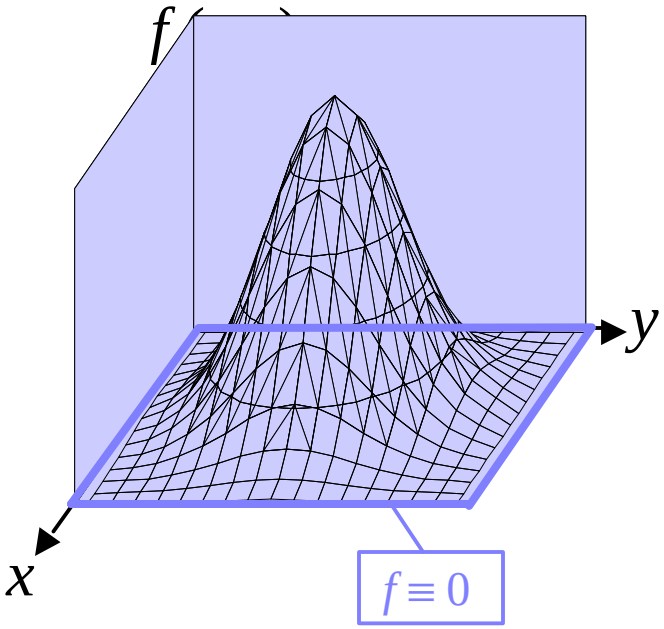
`Dim`

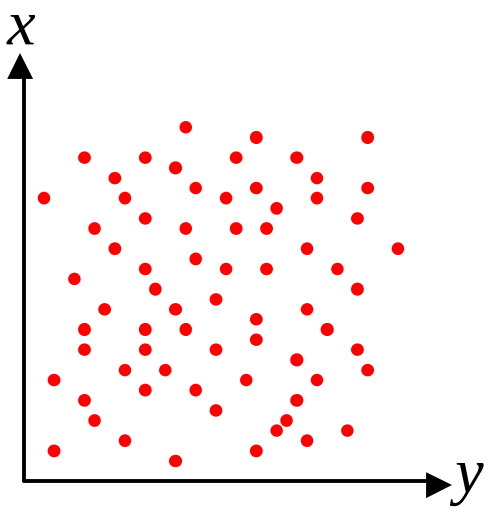


```
template<int Dim, class T, class EngineTag>
class Array;
```

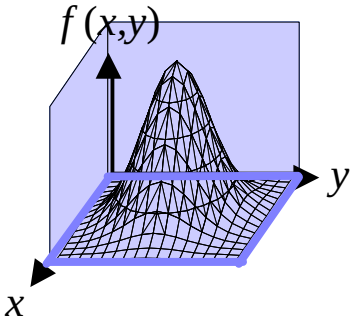
POOMA 2.1







Array **Field** **Particles**

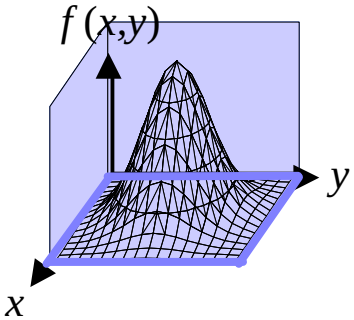


Field Class

```
double
int
Tensor<3,double>
Vector<2,double>
...
```

```
template<class Geometry, class T, class EngineTag>
class Field;
```

```
Brick
MultiPatch<GridTag, CompressibleBrick>
FieldStencilEngine<>
...
```



Field Class (cont'd)

```
template<class Geometry, class T, class EngineTag>
class Field;
```

```
DiscreteGeometry<Cell, RectilinearMesh<3> >
DiscreteGeometry<FaceRCTag<0>, RectilinearMesh<2> >
...
```

```
template<class Centering, class Mesh>
class DiscreteGeometry;
```

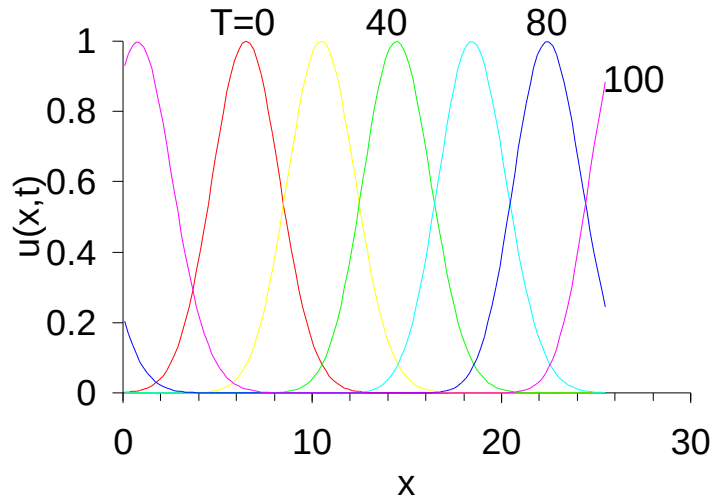
```
template<int Dim, class CoordinateSystem, class T>
class RectilinearMesh;
```

Example: Scalar Advection

$$\frac{d}{dt}u(\vec{x},t) = -\nabla \cdot \vec{F}(\vec{x},t)$$

$$\vec{F}(\vec{x},t) \equiv \vec{v}u(\vec{x},t),$$

solution $\quad u(\vec{x},t) = u(\vec{x} - \vec{v}t, 0)$



$$\left(u_{ijk}^{n+1} - u_{ijk}^{n-1}\right)/(2dt) = -div\left(\vec{v}u_{ijk}^{n}\right)$$

$$= -v_x\left(u_{i+1,j,k}^{n} - u_{i-1,j,k}^{n}\right)/(2dx) - \cdots$$

```
u = uPrev - 2*dt*div<Cell>(v*u);
```

ScalarAdvection.cpp (1/6)

```
#include "Pooma/Fields.h"
int main(int argc, char *argv[])
{
  Pooma::initialize(argc,argv); // Set up the library

  // Create the physical Domains:
  const int Dim    = 1;         // Dimensionality
  const int nVerts = 129;
  const int nCells = nVerts - 1;
  Interval<Dim> vertexDomain;
  for (int d = 0; d < Dim; d++) {
    vertexDomain[d] = Interval<1>(nVerts);
  }
```

ScalarAdvection.cpp (2/6)

```
// Create the (uniform, logically rectilinear) mesh.
Vector<Dim> origin(0.0), spacings(0.2);
typedef UniformRectilinearMesh<Dim> Mesh_t;
Mesh_t mesh(vertexDomain, origin, spacings);


// Create two geometry objects, one with 1 guard layer for
// stencil width; one with none for temporaries:
typedef DiscreteGeometry<Cell, Mesh_t> Geometry_t;
Geometry_t geom(mesh, GuardLayers<Dim>(1));
Geometry_t geomNG(mesh);
```

ScalarAdvection.cpp (3/6)

```
// Create the Fields:

// The flow Field u(x,t):
Field<Geometry_t> u(geom);
// The same, stored at the previous timestep for staggered
// leapfrog plus a useful temporary:
Field<Geometry_t> uPrev(geomNG), uTemp(geomNG);

// Initialize Field to zero everywhere, even global guard layers:
u.all() = 0.0;

// Set up periodic boundary conditions on all mesh faces:
u.addBoundaryConditions(AllPeriodicFaceBC());
```

ScalarAdvection.cpp (4/6)

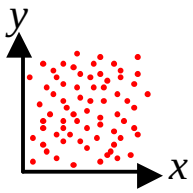
```
// Initial condition u(x,0), symmetric pulse about nCells/4:
const double pulseWidth = spacings(0)*nCells/8;
Loc<Dim> pulseCenter;
for (int d = 0; d < Dim; d++)
  { pulseCenter[d] = Loc<1>(nCells/4); }

Vector<Dim> u0 = u.x(pulseCenter);

u = exp(-dot(u.x() - u0, u.x() - u0) / (2.0 * pulseWidth));

// Output the initial field on its physical domain:
std::cout << "Time = 0:\n" << u() << std::endl;

const Vector<Dim> v(0.2);   // Propagation velocity
const double dt = 0.1;      // Timestep
```

ScalarAdvection.cpp (5/6)

```
// Prime the leapfrog from initial conditions:
uPrev = u;

// Preliminary forward Euler timestep, using canned POOMA
// stencil-based operator div() for the spatial difference:
u -= dt * div<Cell>(v * u);

// Staggered leapfrog timestepping. Spatial derivative is
// again canned POOMA operator div():
for (int timestep = 2; timestep <= 1000; timestep++)
{
  uTemp = u;
  u = uPrev - 2.0 * dt * div<Cell>(v * u);
  uPrev = uTemp;
}
```

ScalarAdvection.cpp (6/6)

```
  Pooma::finalize();
  return 0;
}
```

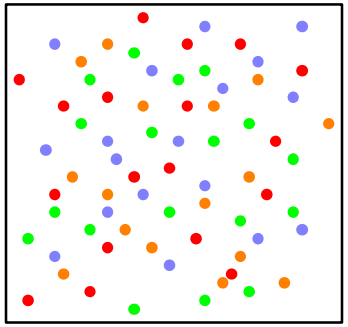


`Particles` Class

```
template<class PTraits> class Particles;
```

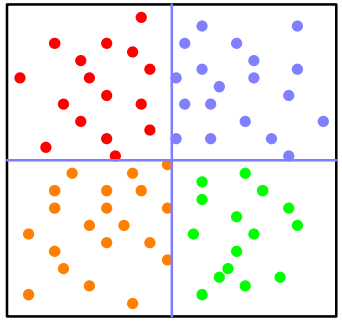
- Container of particle *attributes*:
  - Special 1D array type allows insertion/deletion of elements
    `template<class T, class EngineTag> DynamicArray;`
  - **`PTraits`** defines
    - **`EngineTag`** for attributes
    - Particle layout type for multi-patch **`DynamicArray`**s

`UniformLayout`



Patch 0,1,2,3

`SpatialLayout`



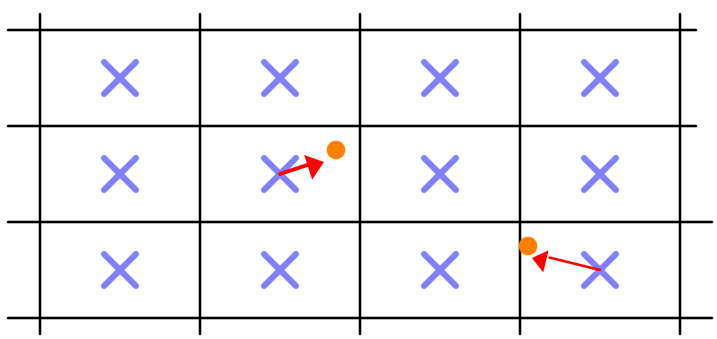
Example: Particle-In-Cell

Particles of charge q in electric field $\vec{E}(\vec{x})$

Positions, velocities $\{\vec{x}_i, \vec{v}_i : i = 1, \ldots N\}$

$$\frac{d\vec{x}_i}{dt} = \vec{v}_i \qquad \frac{d\vec{v}_i}{dt} = \frac{q}{m}\vec{E}(\vec{x}_i)$$

- ***E*** discretized on mesh as `Field` (`Cell` centering)
  - Interpolate to particle positions
    - Nearest-grid-point formula



PIC.cpp (1/9)

```
// Traits class for Particles object
template <class MeshType, class FL>
class CPTraits
{
public:
  // The type of engine to use in the attributes
  typedef MultiPatch<GridTag, Brick> AttributeEngineTag_t;

  // The type of particle layout to use
  typedef SpatialLayout<DiscreteGeometry<Cell, MeshType>, FL>
    ParticleLayout_t;
};
```

PIC.cpp (2/9)

```
// Particles subclass with position, velocity, E-field
template <class PT>
class ChargedParticles : public Particles<PT>
{
public:
  // Typedefs
  typedef          Particles<PT>                          Base_t;
  typedef typename Base_t::AttributeEngineTag_t     EngineTag_t;
  typedef typename Base_t::ParticleLayout_t    ParticleLayout_t;
  typedef typename ParticleLayout_t::AxisType_t      AxisType_t;
  typedef typename ParticleLayout_t::PointType_t    PointType_t;
```

PIC.cpp (3/9)

```
  // Constructor: set up layouts, register attributes
  ChargedParticles(const ParticleLayout_t &pl, double qmi = 1.0)
    : Particles<PT>(pl), qm(qmi)
  {
    addAttribute(R); // Position
    addAttribute(V); // Velocity
    addAttribute(E); // Local electric Field value
  }

  // Position, velocity, local E attributes (as public members)
  DynamicArray<PointType_t, EngineTag_t> R;
  DynamicArray<PointType_t, EngineTag_t> V;
  DynamicArray<PointType_t, EngineTag_t> E;

  double qm; // Charge-to-mass ratio; same for all particles
};
```

PIC.cpp (4/9)

```
// Main simulation routine
int main(int argc, char *argv[])
{
  Pooma::initialize(argc, argv); // Set up the library
  Inform out(NULL,0);            // Thread-0 output stream

  // Create the physical Domains:
  const int Dim    = 2;           // Dimensionality
  const int nVerts = 201;
  const int nCells = nVerts - 1;
  Interval<Dim> vertexDomain;
  for (int d = 0; d < Dim; d++) {
    vertexDomain[d] = Interval<1>(nVerts);
  }
```

PIC.cpp (5/9)

```
// Mesh and geometry objects for cell-centered Field:
typedef UniformRectilinearMesh<Dim> Mesh_t;
Mesh_t mesh(vertexDomain);
typedef DiscreteGeometry<Cell,Mesh_t> Geometry_t;
Geometry_t geometry(mesh);

// Create the electric Field; 4 x 4 x ... decomposition:
typedef MultiPatch<UniformTag, Brick> EngineTag_t;
typedef UniformGridLayout<Dim> FLayout_t;
Loc<Dim> patches(4);
FLayout_t flayout(geometry.physicalDomain(), patches);
Field<Geometry_t, Vector<Dim>, EngineTag_t>
  E(geometry, flayout);

// Initialize (constant) electric field as sin(x):
double E0 = 0.01 * nCells; double pi = acos(-1.0);
E = E0 * sin(2.0*pi * E.x().comp(0) / nCells);
```

PIC.cpp (6/9)

```
// Create a particle layout object for our use
typedef CPTraits<Mesh_t, FLayout_t> PTraits_t;
PTraits_t::ParticleLayout_t layout(geometry, flayout);

// Create Particles object, set periodic boundary conditions
typedef ChargedParticles<PTraits_t> Particles_t;
Particles_t P(layout, 1.0); // Charge-to-mass ratio = 1.0
Particles_t::PointType_t lower(0.0), upper(nCells);
PeriodicBC<Particles_t::PointType_t> bc(lower, upper);
P.addBoundaryCondition(P.R, bc);

// Create an equal number of particles on each patch:
const int NumPart = 400;  // Global number of particles
P.globalCreate(NumPart);
```

PIC.cpp (7/9)

```
// Random initial particle positions, zero velocities.
P.V = Particles_t::PointType_t(0.0);
srand(12345U);
for (int i = 0; i < NumPart; ++i) {
  for (int d = 0; d < Dim; d++) {
    P.R.comp(d)(i) = nCells * rand() /
      static_cast<Particles_t::AxisType_t>(RAND_MAX);
  }
}

// Redistribute particle data based on spatial layout
P.swap(P.R);

const double dt = 1.0; // Timestep
```

PIC.cpp (8/9)

```
// Begin main timestep loop
for (int timestep = 1; timestep <= 20; timestep++) {

  // Advance particle positions
  P.R = P.R + dt * P.V;

  // Apply boundary conditions, update particle distribution
  P.sync(P.R);

  // Gather E field to particle positions
  gather( P.E, E, P.R, NGP() );

  // Advance particle velocities
  P.V = P.V + dt * P.qm * P.E;
}
```

PIC.cpp (9/9)

```
  // Display the final particle positions & velocities
  out << "PIC timestep loop complete." << std::endl;
  out << "---------------------------" << std::endl;
  out << "Final particle data:" << std::endl;
  out << "Particle positions: "  << P.R << std::endl;
  out << "Particle velocities: " << P.V << std::endl;

  Pooma::finalize();
  return 0;
}
```

POOMA R1 Capabilities Missing in 2.1

- Parallel file I/O
  - *Plan: POOMA 2.2 (Nov.), HDF5-based*
- FFT
  - *Plan: POOMA 2.3*
- Cross-box parallelism
  - *Plan: POOMA 2.3*
- Field/Array iterators
  - *Plan: external iterators, flattening Engines (flatten ND to 1D view)*
- Parallel random number generators
  - *Plan: POOMA 2.3*

POOMA 2.1 Capabilities Not in R1

- Array syntax on local patch views of **`Array`**, **`Field`**, **`DynamicArray`**
- Easy user-defined boundary conditions
- Automated way to plug “scalar code” into expressions: **`Stencil`**, **`FieldStencil`**, **`UserFunction`**
- Dynamic adding of attributes to **`Particles`** object

POOMA 2.1 Domain Classes

- Integer-based (discrete array indexing)
  - `Interval<2> I(0, 13, 2, 20)`
    - `[0:13:1, 2:20:1]`
  - `Range<1> R(0, 10, 2)`
    - `[0:10:2]`
  - `Loc<3> L(i1, i2, i3)`
    - `(i1, i2, i3)` scalar index

- Floating-point-based (continuous regions)
  - `Region<2> box(0.0, 1.0, 2.0, 4.0)`
    - ([0.0,1.0], [2.0,4.0])

POOMA 2.1 Domain Classes (cont'd)

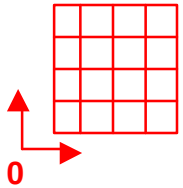
- Array syntax

```
Interval<1> I(0, 13), J(2, 20);
Interval<2> I2(I, J);
Array<2, ....> A(...), B(...), C(...), D(...);
A(I,J) += B(I,J);
C(I2) = B(I2) + pow(D(I2), 3);

// Stencil:
A(I,J) = (A(I+1, J+1) - A(I-1, J-1))/2;

// Equivalent Stencil:
Loc<2> offsets(1,1);
A(I2) = (A(I2 + offsets) - A(I2 - offsets))/2;
```

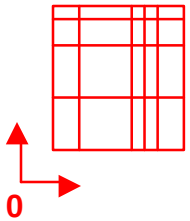
POOMA 2.1 Mesh Classes

- Logically rectilinear

```
template<int Dim, class CoordinateSystem, class T>
class UniformRectilinearMesh;
```




```
template<int Dim, class CoordinateSystem, class T>
class RectilinearMesh;
```



- Example services (member fcns)

```
Array<Dim,PointType_t,PositionFunctor_t> &vertexPositions()
```

  - Compute-based Engine (no storage)
  - Indexable (it's an **`Array`**)

```
Loc<Dim> &cellContaining(PointType_t &pt)
```

  - Finds array index of cell containing pt

POOMA 2.1 Coordinate Systems

- Rectilinear

```
template<int Dim> class Cartesian;
```

- Curvilinear (cylindrical)

```
class Cylindrical;
class Polar
class RhoZ;
class Rho;
```

- Example services:

```
template<class T>
Cylindrical::Volume(Region<3,T> &region);

template<class T>
T Cartesian<Dim>::distance(const Vector<Dim, T> &pt1,
                           const Vector<Dim, T> &pt2);
```

POOMA 2.1 Geometry Classes

- Only one
  - Discrete
  - Centering points relative to a Mesh

```
template<class Centering, class Mesh>
class DiscreteGeometry;
```

    - Partial specializations for **`[Uniform]RectilinearMesh`**

- Example services

```
PositionArray_t &x()
```

    - Compute-based Engine (no storage) for **`[Uniform]RectilinearMesh`**
    - Indexable (it's an **`Array`**)

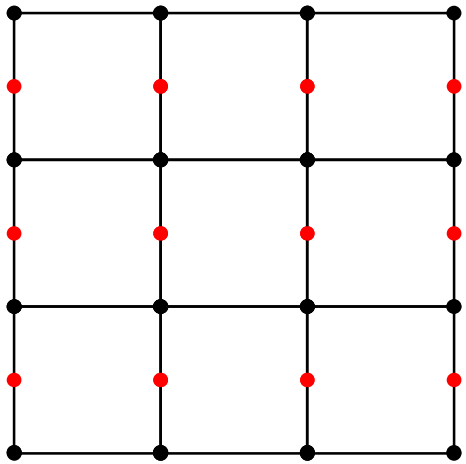
```
Domain_t &physicalDomain()
```

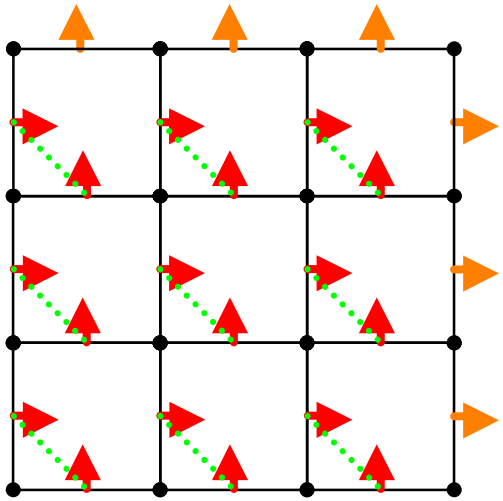
    - Array domain for a **`Field`** on this geometry

POOMA 2.1 Centering Classes

- Centerings with respect to logically rectilinear meshes

```
RectilinearCentering<2, FaceRCTag<0>>
```





```
RectilinearCentering<2,VectorFaceRCTag<2>>
```

`Field` Class Services

- **`Field`** member functions
  - **`operator()`** - Indexing with **`int`**s, **`Interval<>`**, *etc.*
  - **`x()`**- **`Array`** of position vectors
    - Forwards to **`Field::geometry().x()`**
  - **`all()`** - Array view of total index domain
  - **`template<class Category>`**
    `addBoundaryConditions(Category &bc);`
    - Add to **`Field`**'s list of automatic boundary conditions

POOMA 2.1 **`Field`** Boundary Conditions

- Canned types

```
PeriodicFaceBC
LinearExtrapolateFaceBC
ConstantFaceBC<T> , ZeroFaceBC<T>
NegReflectFaceBC, PosReflectFaceBC
```

- User-defined: partial specialization of

```
template<class Subject, class Category> class BCond;
```

  - `class MyBC : public BCondCategory<MyBC>;`
  - Choose class for **`Subject`** parameter, such as **`Field<>`**
  -
```
class BCond<Field<...>, MyBC> {
Field<...>::Domain_t destDomain[,srcDomain];
void applyBoundaryCondition() {...} ...};
```

- Componentwise BC (for multicomponent element types)
  -
```
template<int Dim, class Category>
class ComponentBC;
```

POOMA 2.1 Tiny Types

- Multicomponent

Storage type.
POOMA 2.1:
only **Full**

Compile-time sizes

```
template<int Dim, class T, class EngineTag>
class Vector;
```

Post POOMA 2.1:
Full, Symmetric, Antisymmetric, Diagonal

```
template<int Dim, class T, class EngineTag>
class Tensor;

template<int D1, int D2, class T, class EngineTag>
class TinyMatrix;
```

FieldStencil

```
template<class Functor> class FieldStencil;
```

- **`Functor::operator(i0,i1,...)`** methods
  - scalar code to implement stencil

May be an expression.

- **`FieldStencil<Functor>::operator()(Field<G,T,E>&)`**

Returns **`Field<G2,T2,`** *special compute-based engine***`>`**
Inlines into expressions.

- Canned **`Div<>()`**, **`grad<>()`**, **`average<>()`** functions use **`Div<>`**, **`Grad<>`**, **`Average<>`** **`FieldStencils`** internally.

`Particles` Class Services

- `Particles` member functions
  - `create(...),destroy(...)`

    Allocate/delete elements in `DynamicArray`s for all attributes
  - `template<class Subject, class Object, class BCType>`

    `void addBoundaryConditions(Subject &s, Object &o,`

    `Bctype &b);`

    Add to list of automatic boundary conditions
  - `template<class T> void swap(T &attrib)`

    Reassign patch ownership based on `attrib` values

POOMA 2.1 **`Particles`** Boundary Conditions

- Really *filters*: Out-of-range values of one attribute reset values of another.
  - **`Subject`** = **`Object`** = position attribute
    → conventional boundary condition.
    - Canned BC

```
AbsorbBC<T>, KillBC<T>
PeriodicBC<T>
ReflectBC<T>, ReverseBC<T>
```

- User-defined: partial specialization of

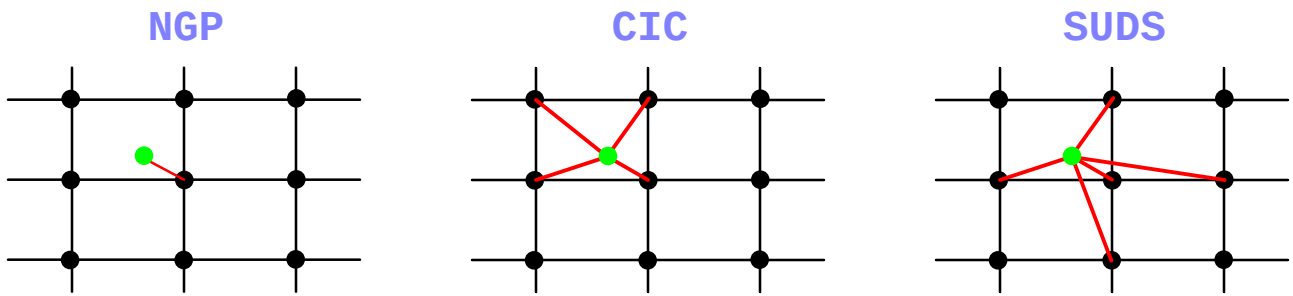
```
template<class Subject, class Object, class BCType>
  class ParticleBC;
```

  - `class MyBC : public ParticleBCType<MyBC>;`
  - Choose class for **`Subject`**, **`Object`** parameters, such as **`DynamicArray<>`**
  - `class ParticleBC<DynamicArray<...>, DynamicArray<>, MyBC>`

```
{
  Subject_t subject_m; Object_t object_m;
  void applyBoundaryCondition(...);
```

Particle-Field Interpolators

- Canned interpolation schemes



- Gather/scatter functions

`gather(`*attribute*, *Field*, *position attribute*, *scheme*`);**

`scatter(`*attribute*, *Field*, *position attribute*, *scheme*`);**

`gatherValue(`*value*, *Field*, *position attribute*, *scheme*`);**

`gatherCache(`*attribute*, *Field*, *position attribute*, *cache*, *scheme*`);**

`gatherCache(`*attribute*, *Field*, *cache*, *scheme*`);**