Finite Element Assembly on Arbitrary Meshes

Matthew Knepley

Computation Institute
University of Chicago
Department of Molecular Biology and Physiology
Rush University Medical Center

Department of Applied Mathematics and Computational Science
King Abdullah University of Science and Technology
Apr 5, 2010
Collaborators

 Automated FEM
  - Andy Terrel (UT Austin)
  - Ridgway Scott (UChicago)
  - Rob Kirby (Texas Tech)

 Sieve
  - Dmitry Karpeev (ANL)
  - Peter Brune (UChicago)
  - Anders Logg (Simula)

 PyLith
  - Brad Aagaard (USGS)
  - Charles Williams (NZ)
Outline

1. Introduction

2. Mesh Distribution
   - Sieve
   - Section
   - Completion
   - Distribution
   - Interfaces

3. Unifying Paradigm

4. Finite Element Assembly
Rethinking meshes produces a simple FEM interface and good code reuse.
Rethinking meshes produces a simple FEM interface and good code reuse.
Rethinking meshes produces a simple FEM interface and good code reuse.
Problems

The biggest problem in scientific computing is **programmability**:

- Lack of usable implementations of modern algorithms
  - Unstructured Multigrid
  - Fast Multipole Method
- Lack of comparison among classes of algorithms
  - Meshes
  - Discretizations

We should reorient thinking from

- characterizing the solution (FEM)
  - “what is the convergence rate (in $h$) of this finite element?”

to

- characterizing the computation (FErari)
  - “how many digits of accuracy per flop for this finite element?”
The biggest problem in scientific computing is \textit{programmability}:

- Lack of widespread implementations of modern algorithms
  - Unstructured Multigrid
  - Fast Multipole Method
- Lack of comparison among classes of algorithms
  - Meshes
  - Discretizations

We should reorient thinking from

- characterizing the solution (FEM)
  - “what is the convergence rate (in $h$) of this finite element?”

to

- characterizing the computation (FErari)
  - “how many digits of accuracy per flop for this finite element?”
Mesh Distribution

- Sieve
  - Section
  - Completion
  - Distribution
  - Interfaces
Sieve is an interface for

- general topologies
- functions over these topologies (bundles)
- traversals

One relation handles all hierarchy

- Vast reduction in complexity
  - Dimension independent code
  - A single communication routine to optimize

- Expansion of capabilities
  - Partitioning and distribution
  - Hybrid meshes
  - Complicated structures and embedded boundaries
  - Unstructured multigrid
Mesh Databases

- "Most" Mesh Libraries
  - Specific geometry
  - Strange constraints
  - Complex query model

Topological Mesh DB
- Single model
- Simple query model
- Can tune implementation

\[ a \]

Lawler, Kalé

\[ a \] Lawler, Kalé
Mesh Databases

• "Most"
  • Specific geometry
  • Strange constraints
  • Complex query model
  • Topological mesh DB
  • Single model
  • Simple query model
  • Can tune implementation

\[^{a}\text{Aagaard, Knepley, Williams}\]

\[^{a}\text{M. Knepley (UC)}\]
Doublet Mesh

- Incidence/covering arrows
  - $\text{cone}(0) = \{2, 3, 4\}$
  - $\text{support}(7) = \{2, 3\}$
Doublet Mesh

- Incidence/covering arrows
- $\text{cone}(0) = \{2, 3, 4\}$
- $\text{support}(7) = \{2, 3\}$
Doublet Mesh

- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$
**Doublet Mesh**

- **Incidence/covering arrows**
- \( \text{closure}(0) = \{0, 2, 3, 4, 7, 8, 9\} \)
- \( \text{star}(7) = \{7, 2, 3, 0\} \)
Doublet Mesh

- Incidence/covering arrows
- $\text{closure}(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $\text{star}(7) = \{7, 2, 3, 0\}$
Doublet Mesh

Incidence/covering arrows

\( \text{meet}(0, 1) = \{4\} \)

\( \text{join}(8, 9) = \{4\} \)
Incidences/covering arrows

$\text{meet}(0, 1) = \{4\}$

$\text{join}(8, 9) = \{4\}$
A Sieve consists of points, and arrows. Each arrow connects a point to another which it covers.

<table>
<thead>
<tr>
<th>cone(p)</th>
<th>sequence of points which cover a given point p</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure(p)</td>
<td>transitive closure of cone</td>
</tr>
<tr>
<td>support(p)</td>
<td>sequence of points which are covered by a given point p</td>
</tr>
<tr>
<td>star(p)</td>
<td>transitive closure of support</td>
</tr>
<tr>
<td>meet(p,q)</td>
<td>minimal separator of closure(p) and closure(q)</td>
</tr>
<tr>
<td>join(p,q)</td>
<td>minimal separator of star(p) and star(q)</td>
</tr>
</tbody>
</table>
The Mesh Dual

Construct mesh dual by reversing sieve arrows and taking the support of each face. Also, take the meet of each cell pair.
Outline

2 Mesh Distribution
- Sieve
- Section
- Completion
- Distribution
- Interfaces
Doublet Section

- **Section interface**
  - $\text{restrict}(0) = \{f_0\}$
  - $\text{restrict}(2) = \{v_0\}$
  - $\text{restrict}(6) = \{e_0, e_1\}$
Doublet Section

Section interface

- \( \text{restrict}(0) = \{ f_0 \} \)
- \( \text{restrict}(2) = \{ v_0 \} \)
- \( \text{restrict}(6) = \{ e_0, e_1 \} \)
**Section interface**

- \( \text{restrict}(0) = \{ f_0 \} \)
- \( \text{restrict}(2) = \{ v_0 \} \)
- \( \text{restrict}(6) = \{ e_0, e_1 \} \)
**Section** interface

- \( \text{restrict}(0) = \{f_0\} \)
- \( \text{restrict}(2) = \{v_0\} \)
- \( \text{restrict}(6) = \{e_0, e_1\} \)
Topological traversals: follow connectivity

- \( \text{restrictClosure}(0) = \{ f_0 e_0 e_1 e_2 e_3 e_4 e_5 v_0 v_1 v_2 \} \)
- \( \text{restrictStar}(7) = \{ v_0 e_0 e_1 e_4 e_5 f_0 \} \)
**Doublet Section**

- **Topological traversals:** follow connectivity
  - \( \text{restrictClosure}(0) = \{ f_0 e_0 e_1 e_2 e_3 e_4 e_5 v_0 v_1 v_2 \} \)
  - \( \text{restrictStar}(7) = \{ v_0 e_0 e_4 e_5 f_0 \} \)
Topological traversals: follow connectivity

- $\text{restrictClosure}(0) = \{ e_0 e_1 e_2 e_3 e_4 e_5 v_0 v_1 v_2 \}$
- $\text{restrictStar}(7) = \{ v_0 e_0 e_1 e_4 e_5 f_0 \}$
Section Definition

Definition

Section is a mapping from sieve points to a vector of values.

<table>
<thead>
<tr>
<th>restrict</th>
<th>return all the values on given subdomain</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>inject subdomain values into global section</td>
</tr>
<tr>
<td>completion</td>
<td>operation to enforce coherence over sieve</td>
</tr>
</tbody>
</table>
2 Mesh Distribution
- Sieve
- Section
- Completion
- Distribution
- Interfaces
Localization
- Restrict to patches (here an edge closure)
- Compute locally
Delta

- Restrict further to the overlap
- Overlap now carries twice the data
Fusion

- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)
Update

- Update local patch data
- Completion = restrict $\rightarrow$ fuse $\rightarrow$ update, in parallel
A ubiquitous parallel form of \(\text{restrict} \rightarrow \text{fuse} \rightarrow \text{update}\)

- Operates on Sections
  - Sieves can be "downcast" to Sections
- Based on two operations
  - Data exchange through overlap
  - Fusion of shared data
Completion has many uses:

**FEM** accumulating integrals on shared faces

**FVM** accumulating fluxes on shared cells

**FDM** setting values on ghost vertices
   - distributing mesh entities after partition
   - redistributing mesh entities and data for load balance
   - accumulating matvec for a partially assembled matrix
Completion has many uses:

**FEM** accumulating integrals on shared faces

**FVM** accumulating fluxes on shared cells

**FDM** setting values on ghost vertices
  - distributing mesh entities after partition
  - redistributing mesh entities and data for load balance
  - accumulating matvec for a partially assembled matrix
Completion has many uses:

- **FEM** accumulating integrals on shared faces
- **FVM** accumulating fluxes on shared cells
- **FDM** setting values on ghost vertices
  - distributing mesh entities after partition
  - redistributing mesh entities and data for load balance
  - accumulating matvec for a partially assembled matrix
Completion has many uses:

**FEM** accumulating integrals on shared faces

**FVM** accumulating fluxes on shared cells

**FDM** setting values on ghost vertices

- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumulating matvec for a partially assembled matrix
Completion has many uses:

**FEM** accumulating integrals on shared faces

**FVM** accumulating fluxes on shared cells

**FDM** setting values on ghost vertices
  - distributing mesh entities after partition
  - redistributing mesh entities and data for load balance
  - accumulating matvec for a partially assembled matrix
Completion has many uses:

**FEM**  accumulating integrals on shared faces

**FVM**  accumulating fluxes on shared cells

**FDM**  setting values on ghost vertices
  - distributing mesh entities after partition
  - redistributing mesh entities and data for load balance
  - accumulating matvec for a partially assembled matrix
Completion has many uses:

**FEM** accumulating integrals on shared faces

**FVM** accumulating fluxes on shared cells

**FDM** setting values on ghost vertices
- distributing mesh entities after partition
- redistributing mesh entities and data for load balance
- accumulating matvec for a partially assembled matrix
Completion can be broken into 4 phases:

1. `restrict()` to an overlap section
2. `copy()` data to the remote overlap section
3. `fuse()` data with existing point data
4. `update()` remote section with fused overlap section data

It is common to combine phases 1 & 2, and also 3 & 4

- Data is moved directly between communication buffers and storage
Section Completion

Mesh Distribution

Completion

2
4
0
3
1
Overlap

Mesh

Process 0

5
17
2
10
7

Process 1

16
2
3
19
12

Overlap

0
1

2
3
4

0
1
2
3
4

M. Knepley (UC)
Section Completion

Mesh Distribution

Mesh

Overlap

Process 0

Process 1

Restrict

5
17
2
10
7

2
7

16
19

16
2
3
19
12

Restrict

0
1
2
3
4

0
1
2
3
4
Section Completion

Mesh Distribution

Completion

Mesh

Process 0

5
17
2
10
7

2
7

2
7

Process 1

16
16
2
3
19
12

Overlap

Copy

Mesh

0
2
1
0
1
4
3
4

Overlap

2
0
1
3
4

M. Knepley (UC)  FEM  KAUST  22 / 89
Section Completion

Mesh Distribution

Completion

M. Knepley (UC)

FEM

KAUST 22 / 89
Section Completion

Mesh Distribution

Completion

22
44
00
3
1
Overlap
3
0
4
2
1
0
2
1
3
4
Mesh
5
17
2
10
7
18
2
6
2
7
26
12
Process 0
Process 1
Update

M. Knepley (UC)
We have a hierarchy of section types of increasing complexity

- **GeneralSection**
  - An arbitrary number of values for each domain point
  - Constrain arbitrary values
  - Atlas is a UniformSection

- **UniformSection**
  - A fixed number of values for each domain point
  - Atlas is a ConstantSection

- **ConstantSection**
  - The same single value for all domain points
  - Only the domain must be completed
Outline

2. Mesh Distribution
   - Sieve
   - Section
   - Completion
   - Distribution
   - Interfaces
Section distribution consists of

- Creation of the local Section
- Distribution of the Atlas (layout Section)
- Completion of the Section
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. Complete() the partition section
   - This distributes the cells
4. Update Overlap with new points
5. Complete() the cone section
   - This distributes the remaining sieve points
6. Update local Sieves
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the partition section
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the cone section
   - This distributes the remaining sieve points
6. Update local Sieves
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. Complete() the partition section
   - This distributes the cells
4. Update Overlap with new points
5. Complete() the cone section
   - This distributes the remaining sieve points
6. Update local Sieves
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. `Complete()` the *partition section*
   - This distributes the cells
4. Update Overlap with new points
5. `Complete()` the *cone section*
   - This distributes the remaining sieve points
6. Update local Sieves
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. Complete() the partition section
   - This distributes the cells
4. Update Overlap with new points
5. Complete() the cone section
   - This distributes the remaining sieve points
6. Update local Sieves
Sieve Distribution

1. Construct local mesh from partition
2. Construct partition overlap
3. Complete() the partition section
   - This distributes the cells
4. Update Overlap with new points
5. Complete() the cone section
   - This distributes the remaining sieve points
6. Update local Sieves
Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of $\text{cone}(\cdot)$'s!
Mesh Distribution

Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of `cone()` s!
Distributing a mesh means

- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of \( \text{cone}() \) s!
Distributing a mesh means
- distributing the topology (Sieve)
- distributing data (Section)

However, a Sieve can be interpreted as a Section of $\text{cone}(\ )$ s!
3rd party packages construct a vertex partition

For FEM, partition dual graph vertices

For FVM, construct hyperpgraph dual with faces as vertices

Assign $\text{closure}(v)$ and $\text{star}(v)$ to same partition
Doublet Mesh Distribution

Mesh Distribution

Distribution

M. Knepley (UC)
FEM
KAUST 29 / 89
Doublet Mesh Distribution

Process 0

Process 1

Mesh Distribution

M. Knepley (UC)
Doublet Mesh Distribution

Mesh Distribution

Distribution

M. Knepley (UC)
2D Example

A simple triangular mesh
2D Example

Sieve for the mesh
Local sieve on process 0
2D Example

Partition Overlap

Process 0

Process 1
Partition Section

3  5  6  7  11  12  13  14  15  16
2D Example

Updated Sieve Overlap
2D Example

Cone Section

13 11 12

11 13 15

15 14 11

16 15 13

3

5

6

7
2D Example

Distributed Sieve

Process 0

Process 1

Mesh Distribution  Distribution

M. Knepley (UC)  FEM  KAUST 30 / 89
2D Example

Coordinate Section

Mesh Distribution Distribution
2D Example

Distributed Coordinate Section

Process 0

<table>
<thead>
<tr>
<th>0.5</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Process 1

<table>
<thead>
<tr>
<th>0.5</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
2D Example

Distributed Mesh
A simple hexahedral mesh
3D Example

Sieve for the mesh

It's complicated!
3D Example

Sieve for the mesh

It's complicated!
3D Example

Partition Overlap

Process 0

Process 1
### 3D Example

**Partition Section**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>24</td>
<td>25</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>89</td>
<td>96</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>107</td>
<td>108</td>
<td>109</td>
</tr>
</tbody>
</table>

---

(M. Knepley (UC))
3D Example

Distributed Mesh

Notice cells are ghosted
2 Mesh Distribution
- Sieve
- Section
- Completion
- Distribution
- Interfaces
Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, …)

- Single relation, covering, handles all hierarchy
  - Rich enough for FEM

- Single operation, completion, for parallelism
  - Enforces consistency of the relation
Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, …)

- Single relation, **covering**, handles all hierarchy
  - Rich enough for FEM

- Single operation, **completion**, for parallelism
  - Enforces consistency of the relation
Sieve Overview

- Hierarchy is the centerpiece
  - Strip out unneeded complexity (dimension, shape, ...)

- Single relation, covering, handles all hierarchy
  - Rich enough for FEM

- Single operation, completion, for parallelism
  - Enforces consistency of the relation
Global and Local

Local (analytical)
- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)
- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)
Global and Local

Local (analytical)
- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
  - Largely dim dependent (e.g. quadrature)

Global (topological)
- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)
Global and Local

Local (analytical)
- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)
- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)
Global and Local

Local (analytical)
- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)
- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)
Global and Local

Local (analytical)
- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions
- Largely dim dependent (e.g. quadrature)

Global (topological)
- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies
- Largely dim independent (e.g. mesh traversal)
Global/Local Dichotomy is the Heart of DD
Software interfaces do not adequately reflect this

- PETSc DA is too specialized
  - Basically 1D methods applied to Cartesian products
- PETSc Index Sets and VecScatters are too fine
  - User “does everything”, no abstraction
- PETSc Linear Algebra (Vec & Mat) is too coarse
  - No access to the underlying connectivity structure
Unstructured Interface (before)

- **Explicit references to element type**
  - `getVertices(edgeID), getVertices(faceID)`
  - `getAdjacency(edgeID, VERTEX)`
  - `getAdjacency(edgeID, dim = 0)`

- **No interface for transitive closure**
  - Awkward nested loops to handle different dimensions

- **Have to recode for meshes with different**
  - dimension
  - shapes
Explicit references to element type
- `getVertices(edgeID)`, `getVertices(faceID)`
- `getAdjacency(edgeID, VERTEX)`
- `getAdjacency(edgeID, dim = 0)`

No interface for transitive closure
- Awkward nested loops to handle different dimensions

Have to recode for meshes with different
- dimension
- shapes
Explicit references to element type
- `getVertices(edgeID), getVertices(faceID)`
- `getAdjacency(edgeID, VERTEX)`
- `getAdjacency(edgeID, dim = 0)`

No interface for transitive closure
- Awkward nested loops to handle different dimensions

Have to recode for meshes with different
- dimension
- shapes
Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, covering, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)

- Simple query set:
  - provides a general API for geometric algorithms
  - leads to simpler implementations
  - can be more easily optimized
Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, **covering**, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)

- Simple query set:
  - provides a general API for geometric algorithms
  - leads to simpler implementations
  - can be more easily optimized
Combinatorial Topology gives us a framework for geometric computing.

- Abstract to a relation, covering, on sieve points
  - Points can represent any mesh element
  - Covering can be thought of as adjacency
  - Relation can be expressed in a DAG (Hasse Diagram)

- Simple query set:
  - Provides a general API for geometric algorithms
  - Leads to simpler implementations
  - Can be more easily optimized
Unstructured Interface (after)

- **NO** explicit references to element type
  - A point may be any mesh element
  - `getCone(point)`: adjacent \((d-1)\)-elements
  - `getSupport(point)`: adjacent \((d+1)\)-elements

- Transitive closure
  - `closure(cell)`: The computational unit for FEM

- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout
Unstructured Interface (after)

- **NO** explicit references to element type
  - A point may be any mesh element
  - `getCone(point)`: adjacent \((d-1)\)-elements
  - `getSupport(point)`: adjacent \((d+1)\)-elements

- Transitive closure
  - `closure(cell)`: The computational unit for FEM

- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout
Unstructured Interface (after)

- **NO** explicit references to element type
  - A point may be any mesh element
  - `getCone(point)`: adjacent `(d-1)`-elements
  - `getSupport(point)`: adjacent `(d+1)`-elements

- Transitive closure
  - `closure(cell)`: The computational unit for FEM

- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout
Unifying Paradigm

Outline

1. Introduction

2. Mesh Distribution

3. Unifying Paradigm
   - DA
   - Mesh
   - DMMG
   - PCFieldSplit

4. Finite Element Assembly
Hierarchy Abstractions

- Generalize to a set of linear spaces
  - **Sieve** provides topology, can also model **Mat**
  - **Section** generalizes **Vec**
  - Spaces interact through an **Overlap** (just a Sieve)

- Basic operations
  - Restriction to finer subspaces, **restrict()*/**update()*
  - Assembly to the subdomain, **complete()**

- Allow reuse of geometric and multilevel algorithms
3 Unifying Paradigm
   - DA
   - Mesh
   - DMMG
   - PCFieldSplit
Residual Evaluation

The **DM** interface is based upon *local* callback functions

- FormFunctionLocal()
- FormJacobianLocal()

Callbacks are registered using

- SNESSetDM(), TSSetDM()
- DMSNESetFunctionLocal(), DMTSSetJacobianLocal()

When PETSc needs to evaluate the nonlinear residual $\mathbf{F}(\mathbf{x})$,

- Each process evaluates the local residual
- PETSc assembles the global residual automatically
  - Uses DMLocalToGlobal() method
Ghost Values

To evaluate a local function $f(x)$, each process requires
- its local portion of the vector $x$
- its *ghost values*, bordering portions of $x$ owned by neighboring processes
DMDA Local Function

User provided function calculates the nonlinear residual (in 2D)

(* lf *)(DMDALocalInfo *info, PetscScalar***x, PetscScalar **r, void *ctx)

**info:** All layout and numbering information

**x:** The current solution (a multidimensional array)

**r:** The residual

**ctx:** The user context passed to DMDASNESSetFunctionLocal()

The local DMDA function is activated by calling

DMDASNESSetFunctionLocal(dm, INSERT_VALUES, lfunc, &ctx)
Bratu Residual Evaluation

\[ \Delta u + \lambda e^u = 0 \]

```c
ResLocal(DMDALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx) {
    for (j = info->ys; j < info->ys+info->ym; ++j) {
        for (i = info->xs; i < info->xs+info->xm; ++i) {
            u = x[j][i];
            if (i==0 || j==0 || i == M || j == N) {
                f[j][i] = 2.0*(hydhx+hxdhy)*u; continue;
            }
            u_xx = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
            u_yy = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
            f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
        }
    }
}
```

$PETSC_DIR/src/snes/examples/tutorials/ex5.c
DMDA Local Jacobian

User provided function calculates the Jacobian (in 2D)

\[
(* \text{ljac} )(\text{DMDALocalInfo } \text{info, PetscScalar**x, Mat J, void *ctx})
\]

**info:** All layout and numbering information  
**x:** The current solution  
**J:** The Jacobian  
**ctx:** The user context passed to DASetLocalJacobian()

The local DMDA function is activated by calling

\[
\text{DMDASNESSetJacobianLocal(dm, ljac, &ctx)}
\]
The **DMDA** object contains only layout (topology) information
- All field data is contained in PETSc **Vecs**

Global vectors are parallel
- Each process stores a unique local portion
- `DMCreateGlobalVector(DM da, Vec *gvec)`

Local vectors are sequential (and usually temporary)
- Each process stores its local portion plus ghost values
- `DMCreateLocalVector(DM da, Vec *lvec)`
- includes ghost and boundary values!
Unifying Paradigm

Updating Ghosts

Two-step process enables overlapping computation and communication

- DMGlobalToLocalBegin(da, gvec, mode, lvec)
  - \texttt{gvec} provides the data
  - \texttt{mode} is either \texttt{INSERT\_VALUES} or \texttt{ADD\_VALUES}
  - \texttt{lvec} holds the local and ghost values
- DMGlobalToLocalEnd(da, gvec, mode, lvec)
  - Finishes the communication

The process can be reversed with DALocalToGlobalBegin/End().
Outline

3 Unifying Paradigm
- DA
- Mesh
- DMMG
- PCFieldSplit
The **DMMesh** interface also uses *local* callback functions

- maps between **global** Vec and **local** Vec
- Local vectors are structured using a **PetscSection**

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Each process evaluates the local residual for each element
- PETSc assembles the global residual automatically
  
  - **DMLocalToGlobal()** works just as in the structured case
Multiple Mesh Types

- Triangular
- Tetrahedral
- Rectangular
- Hexahedral
Outline

3 Unifying Paradigm

- DA
- Mesh
- DMMG
- PCFieldSplit
Multigrid Paradigm

The **DM** interface uses the *local* callback functions to
- assemble global functions/operators from local pieces
- assemble functions/operators on coarse grids

Then **PCMG** organizes
- control flow for the multilevel solve, and
- projection and smoothing operators at each level.
DM supplies global residual and Jacobian to SNES
  - User supplies local version to DM
    - The `Rhs_*()` and `Jac_*()` functions in the example

- Allows automatic parallelism
- Allows grid hierarchy
  - Enables multigrid once interpolation/restriction is defined
- Paradigm is developed in unstructured work
  - Solve needs scatter into contiguous global vectors (initial guess)
- Handle Neumann BC using `KSPSetNullSpace()`
Allows multigrid with some simple command line options

- `-pc_type mg, -pc_mg_levels`
- `-pc_mg_type, -pc_mg_cycle_type, -pc_mg_galerkin`
- `-mg_levels_1_ksp_type, -mg_levels_1_pc_type`
- `-mg_coarse_ksp_type, -mg_coarse_pc_type`
- `-da_refine, -ksp_view`

Interface also works with GAMG and 3rd party packages like ML
Outline

3 Unifying Paradigm
- DA
- Mesh
- DMMG
- PCFieldSplit
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubMatrix()* for efficiency

- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping

- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubMatrix()* for efficiency

- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping

- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

FieldSplit provides the **buildings blocks** for multiphysics preconditioning.
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubMatrix()* for efficiency
- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping
- is composable with ANY PETSc solver and preconditioner
  - This can be done recursively

Notice that this works in exactly the same manner as
- multiple resolutions (MG, FMM, Wavelets)
- multiple domains (Domain Decomposition)
- multiple dimensions (ADI)
Preconditioning

Several varieties of preconditioners can be supported:

- Block Jacobi or Block Gauss-Siedel
- Schur complement
- Block ILU (approximate coupling and Schur complement)
- Dave May’s implementation of Elman-Wathen type PCs
  which only require actions of individual operator blocks

Notice also that we may have any combination of

- “canned” PCs (ILU, AMG)
- PCs needing special information (MG, FMM)
- custom PCs (physics-based preconditioning, Born approximation)

since we have access to an algebraic interface
Outline

1 Introduction

2 Mesh Distribution

3 Unifying Paradigm

4 Finite Element Assembly
   - Layout
   - Integration
   - Assembly
   - Examples
Mathematics Puzzle

- \(-\Delta u + \nabla p = f\)
- \(\nabla \cdot u = 0\)

\[ A x = b \]

Topology

Functional Analysis

Linear Algebra

PDEs
FEM Components

- Section definition
- Integration
- Assembly and Boundary conditions
Outline

4 Finite Element Assembly
  • Layout
  • Integration
  • Assembly
  • Examples
We only need the fiber dimension (# of unknowns) of each sieve point (piece of the mesh)

- Determined by discretization
  - By symmetry, only depend on point depth
  - Obtained from FIAT
  - Modified by BC
  - Decouples storage and parallelism from discretization
Section Allocation

We only need the fiber dimension (# of unknowns) of each sieve point (piece of the mesh)

- Determined by discretization
- By symmetry, only depend on point depth
- Obtained from FIAT
- Modified by BC
- Decouples storage and parallelism from discretization
We only need the fiber dimension (\# of unknowns) of each sieve point (piece of the mesh)

- Determined by discretization
- By symmetry, only depend on point depth
- Obtained from FIAT
- Modified by BC
- Decouples storage and parallelism from discretization
Section Allocation

We only need the fiber dimension (# of unknowns) of each sieve point (piece of the mesh)

- Determined by discretization
- By symmetry, only depend on point depth
- Obtained from FIAT
- Modified by BC

- Decouples storage and parallelism from discretization
Section Allocation

We only need the fiber dimension (# of unknowns) of each sieve point (piece of the mesh)

- Determined by discretization
- By symmetry, only depend on point depth
- Obtained from FIAT
- Modified by BC
- Decouples storage and parallelism from discretization
Outline

4 Finite Element Assembly
- Layout
- Integration
- Assembly
- Examples
Finite Element Integrator And Tabulator by Rob Kirby

http://fenicsproject.org/

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

Can build arbitrary elements by specifying the Ciarlet triple \((K, P, P')\)

FIAT is part of the FEniCS project
Finite Element Integrator And Tabulator by Rob Kirby

http://fenicsproject.org/

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

Can build arbitrary elements by specifying the Ciarlet triple \((K, P, P')\)

FIAT is part of the FEniCS project
FIAT Integration

The `quadrature.fiat` file contains:
- An element (usually a family and degree) defined by FIAT
- A quadrature rule

It is run
- automatically by `make`, or
- independently by the user

It can take arguments
- `-element_family` and `-element_order`, or
- `make` takes variables `ELEMENT` and `ORDER`

Then `make` produces `quadrature.h` with:
- Quadrature points and weights
- Basis function and derivative evaluations at the quadrature points
- Integration against dual basis functions over the cell
- Local dofs for Section allocation
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
- Scalars are invariant
  - Lagrange
- Vectors transform as $J^{-T}$
  - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
- Scalars are invariant
  - Lagrange
- Vectors transform as $J^{-T}$
  - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
  - Scalars are invariant
    - Lagrange
  - Vectors transform as $J^{-T}$
    - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
- Scalars are invariant
  - Lagrange
- Vectors transform as $J^{-T}$
  - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
  - Scalars are invariant
    - Lagrange
  - Vectors transform as $J^{-T}$
    - Hermite
  - Normal vectors require Piola transform and a choice of orientation
    - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
- Scalars are invariant
  - Lagrange
- Vectors transform as $J^{-T}$
  - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
Kinds of Unknowns

We must map local unknowns to the global basis

- FIAT reports the kind of unknown
- Scalars are invariant
  - Lagrange
- Vectors transform as $J^{-T}$
  - Hermite
- Normal vectors require Piola transform and a choice of orientation
  - Raviart-Thomas
- Moments transform as $|J^{-1}|$
  - Nedelec
- May involve a transformation over the entire closure
  - Argyris
- Conjecture by Kirby relates transformation to affine equivalence
- We have not yet automated this step (FFC, Mython)
FFC is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

\[ a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall (\tau, w) \in V \]

where

\[ a((\tau, w), (\sigma, u)) = \int_{\Omega} \tau \sigma - \nabla \cdot \tau u + w \nabla \cdot u \, dx \]

\[ L((\tau, w)) = \int_{\Omega} w f \, dx \]
shape = "triangle"

BDM1 = FiniteElement("Brezzi–Douglas–Marini",shape,1)
DG0 = FiniteElement("Discontinuous Lagrange",shape,0)

element = BDM1 + DG0
(tau, w) = TestFunctions(element)
(sigma, u) = TrialFunctions(element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma))*dx

f = Function(DG0)
L = w*f*dx
Here is a discontinuous Galerkin formulation of the Poisson equation:

\[ a(v, u) = L(v) \quad \forall v \in V \]

where

\[
a(v, u) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \\
+ \sum_{S} \int_{S} - \langle \nabla v \rangle \cdot [u]_{n} - [v]_{n} \cdot \langle \nabla u \rangle - (\alpha/h)vu \, dS \\
+ \int_{\partial \Omega} - \nabla v \cdot [u]_{n} - [v]_{n} \cdot \nabla u - (\gamma/h)vu \, ds \\
L(v) = \int_{\Omega} vf \, dx
\]
Finite Element Assembly
Integration

FFC
DG Poisson

DG1 = FiniteElement("Discontinuous Lagrange",shape,1)
v = TestFunctions(DG1)
u = TrialFunctions(DG1)
f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")
a = dot(grad(v), grad(u)) * dx
   - dot(avg(grad(v)), jump(u, n)) * dS
   - dot(jump(v, n), avg(grad(u))) * dS
   + alpha/h*dot(jump(v, n) + jump(u, n)) * dS
   - dot(grad(v), jump(u, n)) * ds
   - dot(jump(v, n), grad(u)) * ds
   + gamma/h*v*u*ds
L = v*f*dx + v*g*ds
Outline

4 Finite Element Assembly

- Layout
- Integration
- Assembly
- Examples
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    coords = mesh->restrict(coordinates, c);
    v0, J, invJ, detJ = computeGeometry(coords);
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    inputVec = mesh->restrict(U, c);
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q] * detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    // Compute cell geometry
    // Retrieve values from input vector
    for(q = 0; q < numQuadPoints; ++q) {
        // Transform coordinates
        for(f = 0; f < numBasisFuncs; ++f) {
            // Constant term
            // Linear term
            // Nonlinear term
            elemVec[f] *= weight[q] * detJ;
        }
    }
    // Update output vector
}
// Aggregate updates

<Compute cell geometry>
<Retrieve values from input vector>
<Transform coordinates>
<Constant term>
<Linear term>
<Nonlinear term>
elemVec[f] *= weight[q] * detJ;

<Update output vector>
cells = mesh->heightStratum(0);
for (c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for (q = 0; q < numQuadPoints; ++q) {
        realCoords = J*refCoords[q] + v0;
        for (f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            elemVec[f] += basis[q,f]*rhsFunc(realCoords);
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q] * detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            for(d = 0; d < dim; ++d)
                for(e) testDerReal[d] += invJ[e,d]*basisDer[q,f,e];
            for(g = 0; g < numBasisFuncs; ++g) {
                for(d = 0; d < dim; ++d)
                    for(e) basisDerReal[d] += invJ[e,d]*basisDer[q,g,e];
                elemMat[f,g] += testDerReal[d]*basisDerReal[d];
                elemVec[f] += elemMat[f,g]*inputVec[g];
            }
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}

<Aggregate updates>
cells = mesh->heightStratum(0);
for (c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for (q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for (f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      <Nonlinear term>
      elemVec[f] *= weight[q] * detJ;
    }
  }
  <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
  <Compute cell geometry>
  <Retrieve values from input vector>
  for(q = 0; q < numQuadPoints; ++q) {
    <Transform coordinates>
    for(f = 0; f < numBasisFuncs; ++f) {
      <Constant term>
      <Linear term>
      elemVec[f] += basis[q,f]*lambda*exp(inputVec[f])
      elemVec[f] *= weight[q]*detJ;
    }
  }
  <Update output vector>
}

<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q]*detJ;
        }
    }
    <Update output vector>
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for (c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for (q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for (f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q] * detJ;
        }
    }
    mesh->updateAdd(F, c, elemVec);
}
<Aggregate updates>
cells = mesh->heightStratum(0);
for (c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for (q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for (f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q] * detJ;
        }
    }
    <Update output vector>
}

<Aggregate updates>
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) {
    <Compute cell geometry>
    <Retrieve values from input vector>
    for(q = 0; q < numQuadPoints; ++q) {
        <Transform coordinates>
        for(f = 0; f < numBasisFuncs; ++f) {
            <Constant term>
            <Linear term>
            <Nonlinear term>
            elemVec[f] *= weight[q] * detJ;
        }
    }
    <Update output vector>
}
Distribution<Mesh>::completeSection(mesh, F);
**Boundary Conditions**

**Dirichlet** conditions may be expressed as

\[ u|_\Gamma = g \]

and implemented by constraints on dofs in a Section

- The user provides a function.

**Neumann** conditions may be expressed as

\[ \nabla u \cdot \hat{n}|_\Gamma = h \]

and implemented by explicit integration along the boundary

- The user provides a weak form.
Dirichlet Values

- Topological boundary is marked during generation
- Cells bordering boundary are marked using markBoundaryCells()

To set values:
1. Loop over boundary cells
2. Loop over the element closure
3. For each boundary point $i$, apply the functional $N_i$ to the function $g$

The functionals are generated with the quadrature information

Section allocation applies Dirichlet conditions automatically
- Values are stored in the Section
- restrict() behaves normally, update() ignores constraints
We would like the action of a dual basis vector (functional)

\[ \langle \mathcal{N}_i, f \rangle = \int_{\text{ref}} N_i(x) f(x) dV \]

- Projection onto \( \mathcal{P} \)
- Code is generated from FIAT specification
  - Python code generation package inside PETSc
- Common interface for all elements
Assembly with Dirichlet Conditions

The original equation may be partitioned into
- unknowns in the interior ($I$)
- unknowns on the boundary ($\Gamma$)

so that we obtain

\[
\begin{pmatrix}
A_{II} & A_{I\Gamma} \\
A_{\Gamma I} & A_{\Gamma\Gamma}
\end{pmatrix}
\begin{pmatrix}
u_I \\
u_{\Gamma}
\end{pmatrix}
= \begin{pmatrix} f_I \\
f_{\Gamma}
\end{pmatrix}
\]

However $u_{\Gamma}$ is known, so we may reduce this to

\[A_{II}u_I = f_I - A_{I\Gamma}u_{\Gamma}\]

We will show that our scheme automatically constructs this extra term.
Assembly with Dirichlet Conditions

Residual Assembly

\[
\begin{align*}
\mathbf{u} & = \begin{bmatrix} 5 & 1 & 3 & 7 \end{bmatrix} \\
\mathbf{f} & = \begin{bmatrix} 5 & 0 & 0 & 0 \end{bmatrix}
\end{align*}
\]
Assembly with Dirichlet Conditions

Residual Assembly

\[ \begin{align*}
    &u = \begin{bmatrix} 5 & 1 & 3 & 7 \end{bmatrix} \\
    &f = \begin{bmatrix} 5 & 0 & 0 & 0 \end{bmatrix}
\end{align*} \]

Restrict
Assembly with Dirichlet Conditions

Residual Assembly

$$\mathbf{u} = \begin{bmatrix} 5 \\ 1 \\ 3 \\ 7 \end{bmatrix}$$

$$\mathbf{f} = \begin{bmatrix} 5 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Compute

$$\begin{bmatrix} 0.5 & 0.0 & -0.5 \\ 0.0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$
Assembly with Dirichlet Conditions

Residual Assembly

\( \mathbf{u} \)

\[
\begin{bmatrix}
5 & 1 & 3 & 7
\end{bmatrix}
\]

\( \mathbf{f} \)

\[
\begin{bmatrix}
5 & 0 & 0 & 0
\end{bmatrix}
\]

Compute

\[
\begin{pmatrix}
A_{\Gamma\Gamma} & A_{\Gamma\Pi} \\
A_{\Pi\Gamma} & A_{\Pi\Pi}
\end{pmatrix}
\begin{bmatrix}
5 \\
1 \\
3
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
-1 \\
0
\end{bmatrix}
\]

This piece contains rhs values
### Assembly with Dirichlet Conditions

#### Residual Assembly

<table>
<thead>
<tr>
<th>( u )</th>
<th>5</th>
<th>1</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>5</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Update**

\[-1\]

\[0\]
Outline

4 Finite Element Assembly
- Layout
- Integration
- Assembly
- Examples
PyLith

- Multiple problems
  - Dynamic rupture
  - Quasi-static relaxation

- Multiple models
  - Nonlinear visco-plastic
  - Finite deformation
  - Fault constitutive models

- Multiple meshes
  - 1D, 2D, 3D
  - Hex and tet meshes

- Parallel
  - PETSc solvers
  - DMPLex mesh management

\[\text{Aagaard, Knepley, Williams}\]
Multiple Mesh Types

Triangular

Tetrahedral

Rectangular

Hexahedral
Cohesive Cells

Original Mesh

Mesh with Cohesive Cell

Exploded view of meshes
Cohesive cells are used to enforce slip conditions on a fault

- Demand complex mesh manipulation
  - We allow specification of only fault vertices
  - Must “sew” together on output
- Use Lagrange multipliers to enforce constraints
  - Forces illuminate physics
- Allow different fault constitutive models
  - Simplest is enforced slip
  - Now have fault constitutive models
In order to create a fault, the generator provides
- a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented.
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces.
- Sieve code works for
  - any dimension
  - any element shape.
Splitting the Mesh

In order to create a fault, the generator provides:
- a set of fault vertices, or
- a set of fault faces.

Fault vertices, unlike fault faces, must be:
- combined into faces on a fault mesh, and
- oriented.

The fault mesh is used to:
- split vertices along the fault
- introduce prism elements between adjacent fault faces.

Sieve code works for:
- any dimension
- any element shape
Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape
In order to create a fault, the generator provides:
- a set of fault vertices, or
- a set of fault faces.

Fault vertices, unlike fault faces, must be:
- combined into faces on a fault mesh, and
- oriented

The fault mesh is used to:
- split vertices along the fault
- introduce prism elements between adjacent fault faces

Sieve code works for:
- any dimension
- any element shape
Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape
Splitting the Mesh

- In order to create a fault, the generator provides
  - a set of fault vertices, or
  - a set of fault faces.
- Fault vertices, unlike fault faces, must be
  - combined into faces on a fault mesh, and
  - oriented
- The fault mesh is used to
  - split vertices along the fault
  - introduce prism elements between adjacent fault faces
- Sieve code works for
  - any dimension
  - any element shape
In order to create a fault, the generator provides
- a set of fault vertices, or
- a set of fault faces.

Fault vertices, unlike fault faces, must be
- combined into faces on a fault mesh, and
- oriented

The fault mesh is used to
- split vertices along the fault
- introduce prism elements between adjacent fault faces

Sieve code works for
- any dimension
- any element shape
Splitting the Mesh

In order to create a fault, the generator provides
- a set of fault vertices, or
- a set of fault faces.

Fault vertices, unlike fault faces, must be
- combined into faces on a fault mesh, and
- oriented

The fault mesh is used to
- split vertices along the fault
- introduce prism elements between adjacent fault faces

Sieve code works for
- any dimension
- any element shape
Reverse-slip Benchmark
Fracture Mechanics

- Full variational formulation
  - Phase field
  - Linear or Quadratic penalty

- Uses TAO optimization
  - Necessary for linear penalty
  - Backtacking

- No prescribed cracks (movie)
  - Arbitrary crack geometry
  - Arbitrary intersections

- Multiple materials
  - Composite toughness

\(^a\)Bourdin
Fracture Mechanics

1 Bourdin

M. Knepley (UC) FEM KAUST 86 / 89
Conclusions

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Dimension and mesh independent code
  - Complete serial code reuse

- Opportunities for optimization
  - Higher level operations missed by traditional compilers
  - Single communication routine to optimize

- Expansion of capabilities
  - Arbitrary elements
  - Unstructured multigrid
  - Multilevel algorithms
References

- **FEniCS Documentation:**
  http://www.fenics.org/wiki/FEniCS_Project
  - Project documentation
  - Users manuals
  - Repositories, bug tracking
  - Image gallery

- **Publications:**
  http://www.fenics.org/wiki/Related_presentations_and_publications
  - Research and publications that make use of FEniCS

- **PETSc Documentation:**
  http://www.mcs.anl.gov/petsc/docs
  - PETSc Users manual
  - Manual pages
  - Many hyperlinked examples
  - FAQ, Troubleshooting info, installation info, etc.
  - Publication using PETSc
Experimentation is Essential!

Proof is not currently enough to examine solvers