Finite Element Assembly on Arbitrary Meshes

Matthew G Knepley [1] and Andy R Terrel [2]

[1]Mathematics and Computer Science Division
Argonne National Laboratory
[2]Department of Computer Science
University of Chicago

March 12, 2008
SIAM Conference on Parallel Processing for Scientific Computing
Atlanta, Georgia

Outline

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

Go Back to the Math

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

Go Back to the Math

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

Go Back to the Math

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

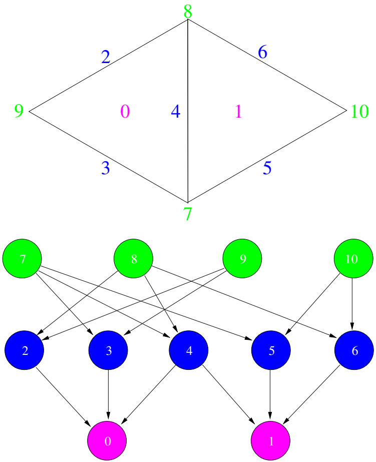
Unstructured Interface (after)

- 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)

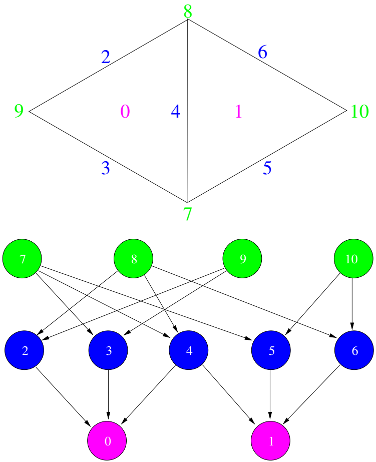
- NO explicit references to element type
  - A point may be any mesh element
  - getCone(point): adjacent (d-1)-elements
  - getSupport(point): adjacent (d+1)-elements
- Transitive closure
  - closure(cell): The computational unit for FEM
- Algorithms independent of mesh
  - dimension
  - shape (even hybrid)
  - global topology
  - data layout

Doublet Mesh



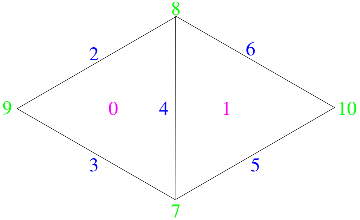
- Incidence/covering arrows
- *cone*(0) = {2, 3, 4}
- *support*(7) = {2, 3}

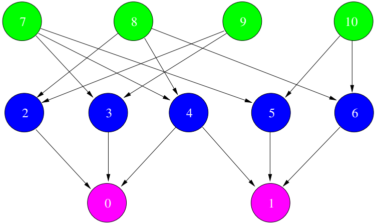
Doublet Mesh



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

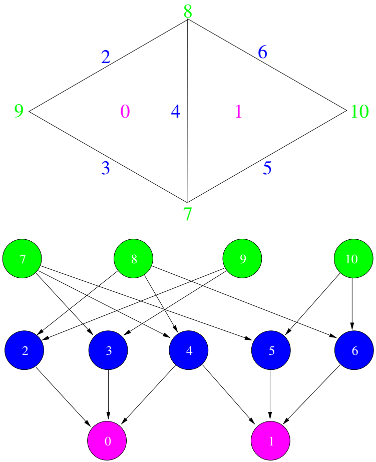
Doublet Mesh





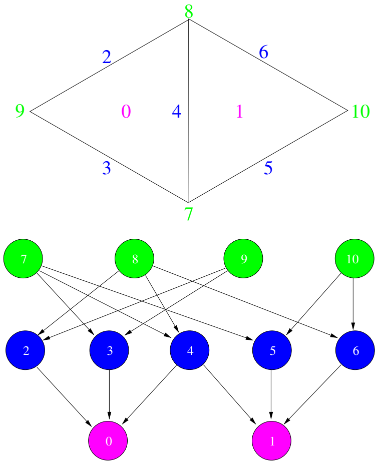
- Incidence/covering arrows
- $cone(0) = \{2, 3, 4\}$
- $support(7) = \{2, 3\}$

Doublet Mesh



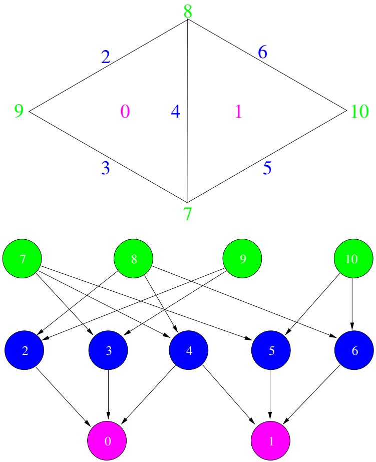
- Incidence/covering arrows
- *closure*(0) = {0, 2, 3, 4, 7, 8, 9}
- *star*(7) = {7, 2, 3, 0}

Doublet Mesh



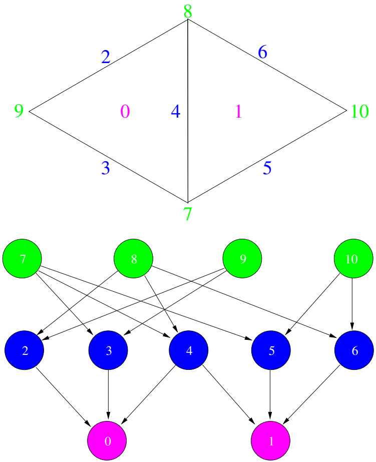
- Incidence/covering arrows
- *closure*(0) = {0, 2, 3, 4, 7, 8, 9}
- *star*(7) = {7, 2, 3, 0}

Doublet Mesh



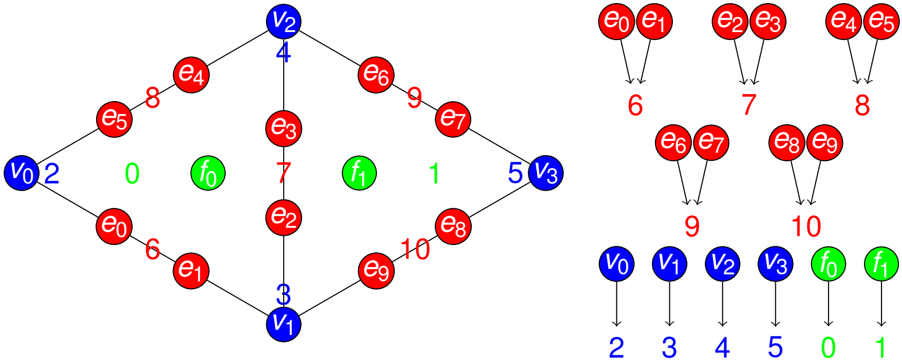
- Incidence/covering arrows
- *meet*$(0, 1) = \{4\}$
- *join*$(8, 9) = \{4\}$

Doublet Mesh



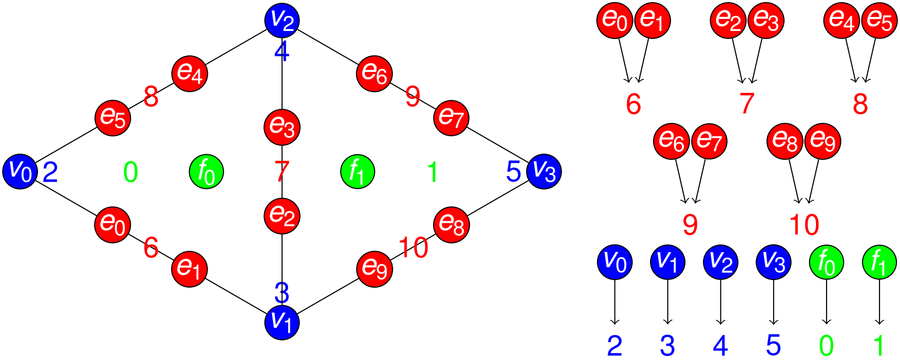
- Incidence/covering arrows
- $meet(0, 1) = \{4\}$
- $join(8, 9) = \{4\}$

Doublet Section



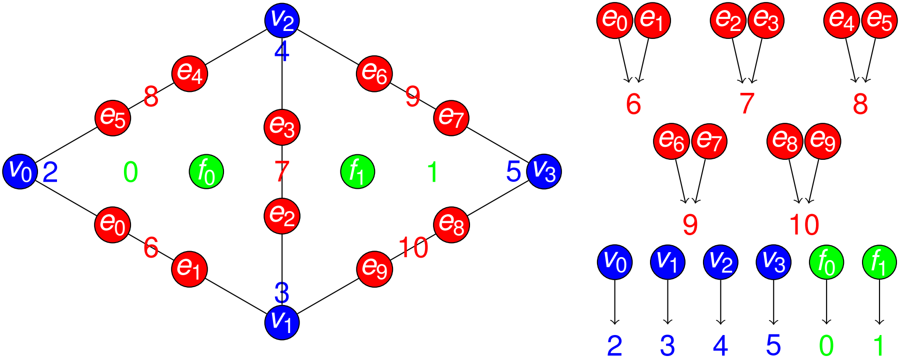
- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

Doublet Section



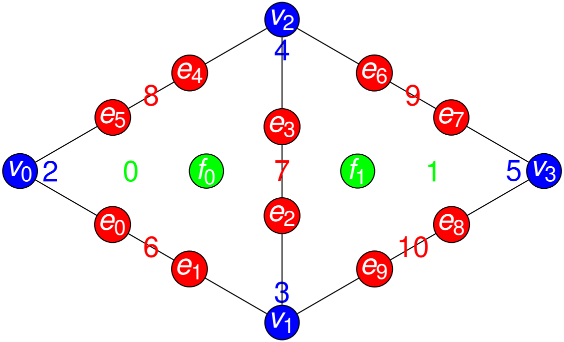
- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

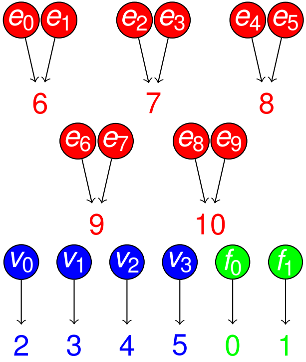
Doublet Section



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

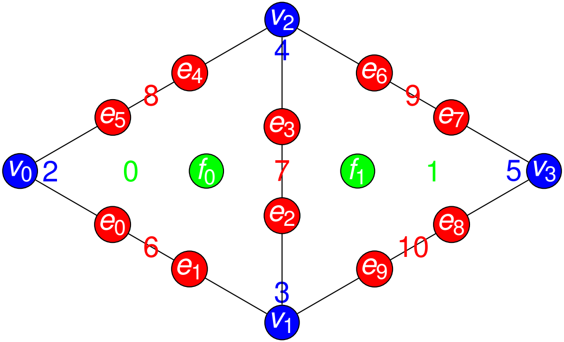
Doublet Section

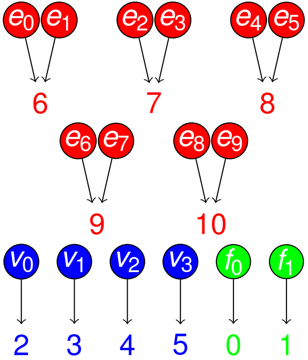




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

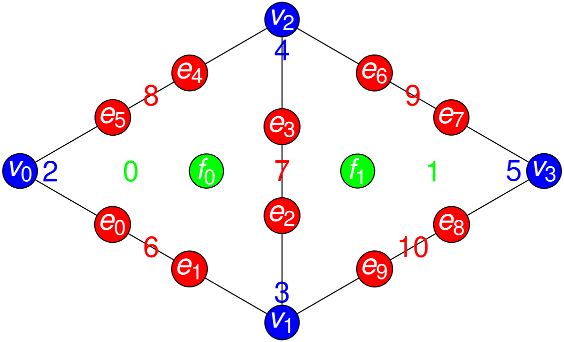
Doublet Section

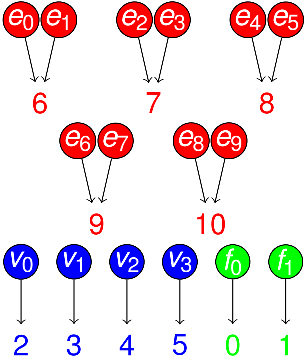




- Topological traversals: follow connectivity
  - $restrictClosure(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $restrictStar(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

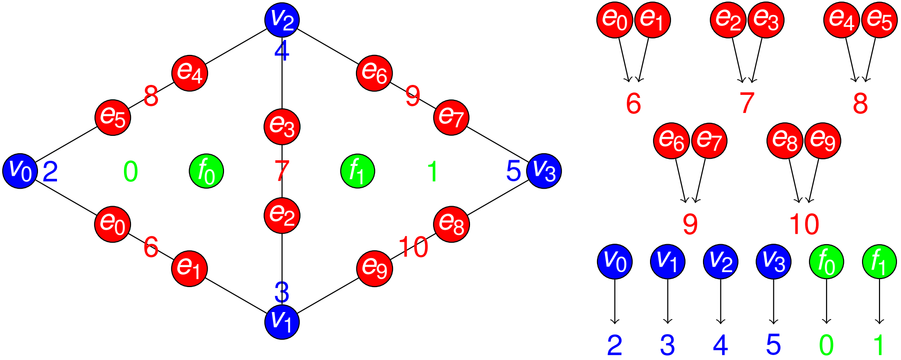
Doublet Section





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

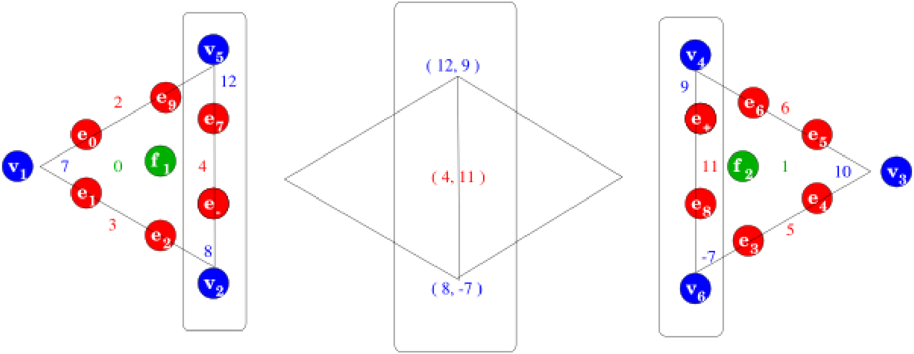
Doublet Section



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

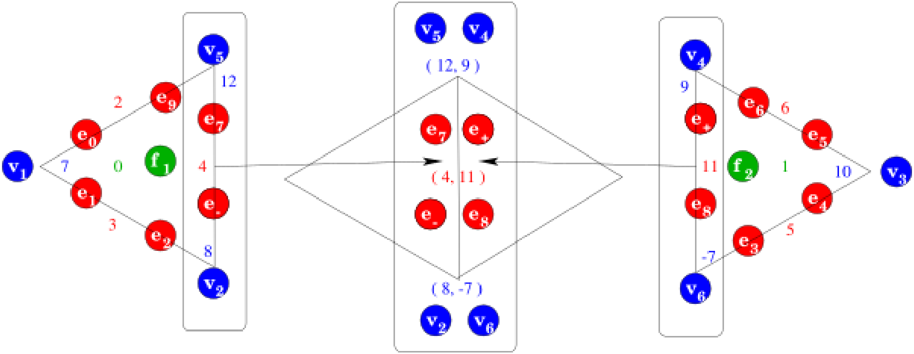
Outline

Restriction



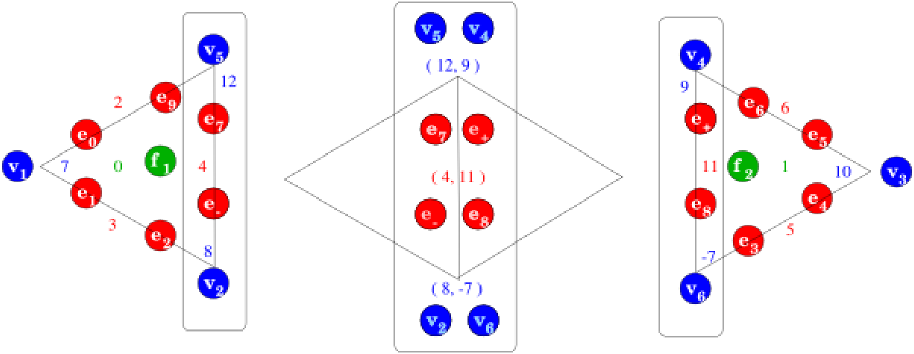
- Localization
  - Restrict to patches (here an edge closure)
  - Compute locally

Delta



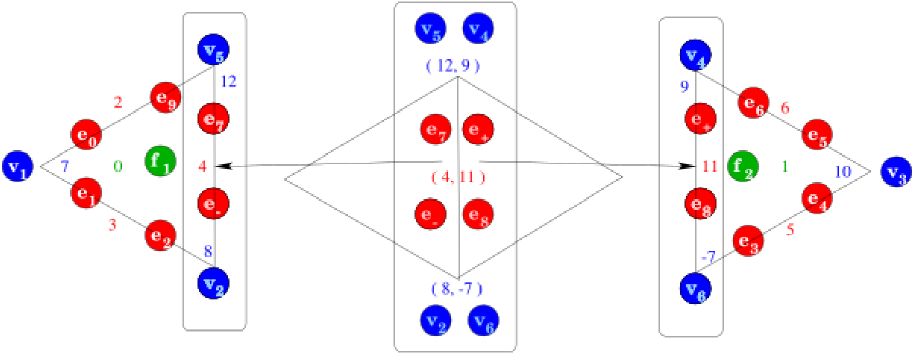
- `Delta`
  - Restrict further to the overlap
  - `Overlap` now carries twice the data

Fusion



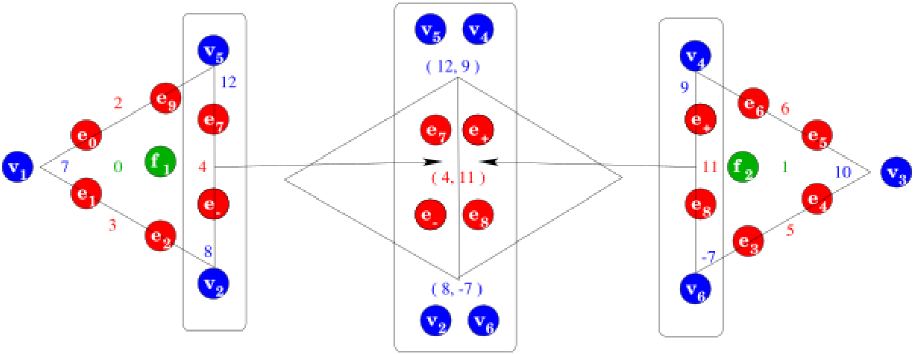
- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)

Update



- `Update`
  - Update local patch data
  - `Completion` = restrict $\longrightarrow$ fuse $\longrightarrow$ update, in parallel

Completion



- A ubiquitous parallel form of *restrict* $\longrightarrow$ *fuse* $\longrightarrow$ *update*
- Operates on Sections
  - Sieves can be "downcast" to Sections
- Based on two operations
  - Data exchange through overlap
  - Fusion of shared data

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

Uses

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
- accumlating matvec for a partially assembled matrix

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!

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!

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!

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!

Outline

FEM Components

- Section definition
- Integration
- Assembly and Boundary conditions

FIAT

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

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

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

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

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

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)

Integration

```
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>
```

Integration

```
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>
}
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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,
      for(g = 0; g < numBasisFuncs; ++g) {
        for(d = 0; d < dim; ++d)
          for(e) basisDerReal[d] += invJ[e,d]*basisDer
          elemMat[f,g] += testDerReal[d]*basisDerReal[
        elemVec[f] += elemMat[f,g]*inputVec[g];
      }
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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>
```

Integration

```
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

Conclusions

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Operate directly at the equation and discretization level
  - Automatic generation of integration/assembly routines
  - Dimension independent code
- Expansion of capabilities
  - Parametric models
  - Optimized implementations of integration
  - Multigrid on arbitrary meshes

Conclusions

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Operate directly at the equation and discretization level
  - Automatic generation of integration/assembly routines
  - Dimension independent code
- Expansion of capabilities
  - Parametric models
  - Optimized implementations of integration
  - Multigrid on arbitrary meshes

Conclusions

Better mathematical abstractions bring concrete benefits

- Vast reduction in complexity
  - Operate directly at the equation and discretization level
  - Automatic generation of integration/assembly routines
  - Dimension independent code
- Expansion of capabilities
  - Parametric models
  - Optimized implementations of integration
  - Multigrid on arbitrary meshes

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

- N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen, *How fast are nonsymmetric matrix iterations?*, SIAM J. Matrix Anal. Appl., **13**, pp.778–795, 1992.
- Anne Greenbaum, Vlastimil Ptak, and Zdenek Strakos, *Any Nonincreasing Convergence Curve is Possible for GMRES*, SIAM J. Matrix Anal. Appl., **17** (3), pp.465–469, 1996.