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
Rethinking the Mesh

Parallelism

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

Transitive closure
- \texttt{closure(cell)}: The computational unit for FEM

Algorithms independent of mesh
- dimension
- shape (even hybrid)
- global topology
- data layout
Rethinking the Mesh

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
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
Incidence/covering arrows

$\text{cone}(0) = \{2, 3, 4\}$

$\text{support}(7) = \{2, 3\}$
Incidence/covering arrows

\[ \text{cone}(0) = \{2, 3, 4\} \]

\[ \text{support}(7) = \{2, 3\} \]
Rethinking the Mesh

Doublet Mesh

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

\[ \text{closure}(0) = \{0, 2, 3, 4, 7, 8, 9\} \]

\[ \text{star}(7) = \{7, 2, 3, 0\} \]
Incidence/covering arrows

$\text{closure}(0) = \{0, 2, 3, 4, 7, 8, 9\}$

$\text{star}(7) = \{7, 2, 3, 0\}$
Incidence/covering arrows

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

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

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

\[ \text{join}(8, 9) = \{4\} \]
**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**

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

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

- restrictClosure(0) = \{ v_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 \}
Topological traversals: follow connectivity

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

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

1. Rethinking the Mesh
2. Parallelism
3. FEM
Localization

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

- Restrict further to the overlap
- Overlap now carries twice the data
- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)
Update

- Update local patch data
- Completion = restrict → fuse → update, in parallel
Parallelism

Completion

- A ubiquitous parallel form of restrict $\rightarrow$ fuse $\rightarrow$ 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
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!
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!
Outline

1. Rethinking the Mesh
2. Parallelism
3. FEM
FEM Components

- Section definition
- Integration
- Assembly and Boundary conditions
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
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)
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>
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>
FEM 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>
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) {
    \texttt{<Compute cell geometry>}
    \texttt{<Retrieve values from input vector>}
    for(q = 0; q < numQuadPoints; ++q) {
        \texttt{<Transform coordinates>}
        for(f = 0; f < numBasisFuncs; ++f) {
            \texttt{<Constant term>}
            for(d = 0; d < \text{dim}; ++d)
                for(e) testDerReal[d] += invJ[e,d]*basisDer[q,e];
            for(g = 0; g < numBasisFuncs; ++g) {
                for(d = 0; d < \text{dim}; ++d)
                    for(e) basisDerReal[d] += invJ[e,d]*basisDer[g,e];
                elemMat[f,g] += testDerReal[d]*basisDerReal[e];
                elemVec[f] += elemMat[f,g]*inputVec[g];
            }
        }
    }
    \texttt{<Nonlinear term>}
    elemVec[f] *= weight[q]*detJ;
}
\texttt{<Update output vector>}
\texttt{<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>
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>
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;
        } // for (f = 0; f < numBasisFuncs; ++f)
    } // for (q = 0; q < numQuadPoints; ++q)
    <Update output vector>
} // for (c = cells->begin(); c != cells->end(); ++c)}
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
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
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
Proof is not currently enough to examine solvers