PETSc and Unstructured Finite Elements

Matthew Knepley and Dmitry Karpeev
Mathematics and Computer Science Division
Argonne National Laboratory

What is PETSc?

- A freely available and supported research code
  - Download from http://www.mcs.anl.gov/petsc
  - Free for everyone, including industrial users
  - Hyperlinked manual, examples, and manual pages for all routines
  - Hundreds of tutorial-style examples
  - Support via email: petsc-maint@mcs.anl.gov
  - Usable from C, C++, Fortran 77/90, and soon Python

What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems

    Cray T3E, SGI Origin, IBM SP, HP 9000, Sub Enterprise
  - Loosely coupled systems, such as networks of workstations

    Compaq,HP, IBM, SGI, Sun, PCs running Linux or Windows
- PETSc History
  - Begun September 1991
  - Over 8,500 downloads since 1995 (version 2), currently 250 per month
- PETSc Funding and Support
  - Department of Energy

    SciDAC, MICS Program
  - National Science Foundation

    CIG, CISE, Multidisciplinary Challenge Program

Linear Algebra Abstractions

- Allows reuse of iterative solvers (Krylov methods)
- **Vec** and **Mat**
- **KSP** and **SNES** can be seen as a nonlinear operator

Hierarchy Abstractions

- Allows reuse of multilevel solvers and preconditioners (Multigrid)
- **DA** is a logically Cartesian grid
  - Also contains linear discretization
  - User works locally and DA handles parallel communication
- **DM** represents a hierarchy of meshes and associated approximation spaces
  - Abstracts the control flow of a multilevel method
  - User can specify local operators, or they are creating using Galerkin
  - User can specify restriction and prologation, or use builtin linear space

Getting More Help

- http://www.mcs.anl.gov/petsc
- Hyperlinked documentation
  - Manual
  - Manual pages for evey method
  - HTML of all example code (linked to manual pages)
- FAQ
- Full support at petsc-maint@mcs.anl.gov
- High profile users
  - David Keyes
  - Rich Martineau
  - Richard Katz

Needs of FEM Simulation

Why Do We Need Another FEM Framework?

- Reusability

  - Rarely goes beyond linear algebra

- Complexity

  - Lack of effective mathematical abstractions creates inpenetrable code

- Extensibility

  - Lack of effective mathematical abstractions prevents generalization

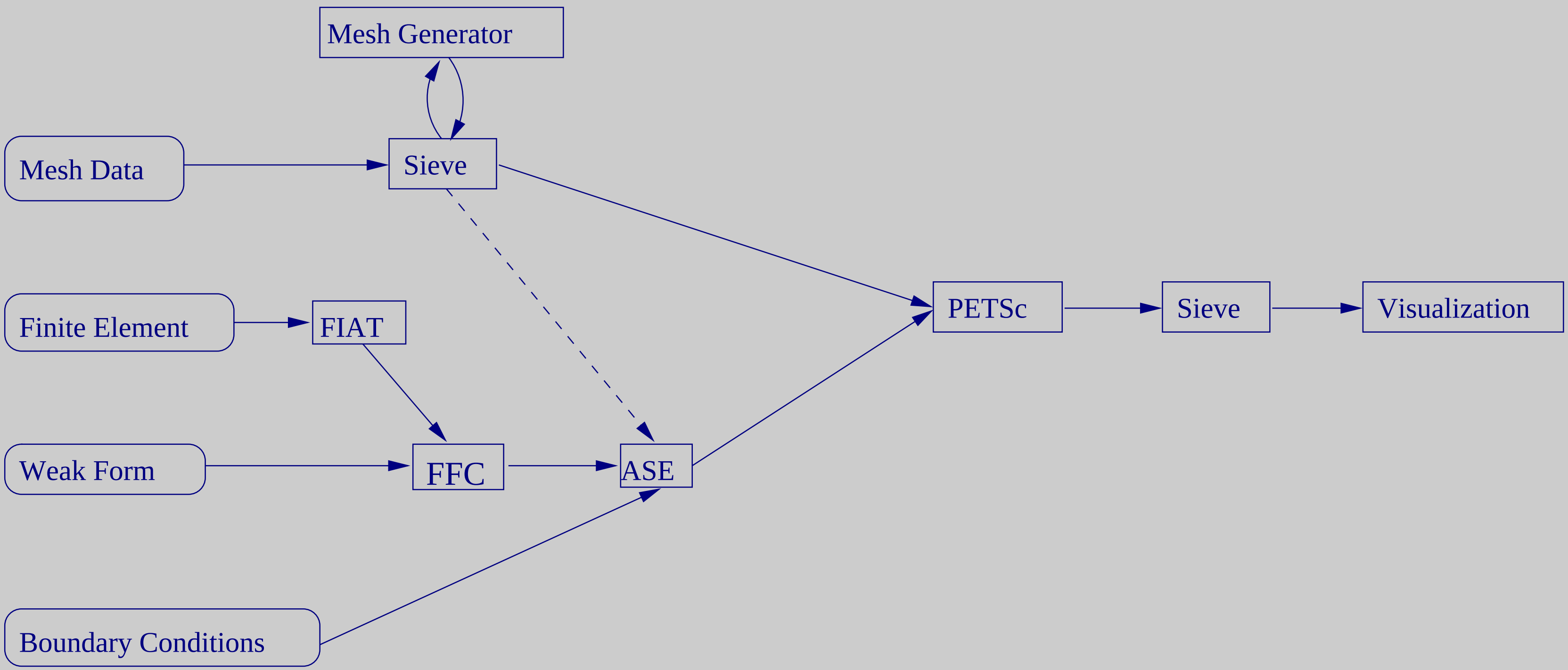
- Modularity

  - Lack of effective mathematical abstractions inhibits component sharing

What Do We Need For FEM Simulation?

- Mesh topology and geometry
  - Hand coded $\Longrightarrow$ Sieve and Array
- Discretization
  - Hand coded $\Longrightarrow$ FIAT
- Weak form PDE
  - Hand coded $\Longrightarrow$ FFC/Expression AST interface
- Integration
  - Hand coded quadrature $\Longrightarrow$ FFC/Generated quadrature
- Assembly
  - A big mess $\Longrightarrow$ Sieve and Array
- Algebraic solve
  - PETSc interfaces
- Preconditioning
  - Often involves LinearAlgebra/Discretization/Mesh

TRIAL FRAMEWORK



The Sieve

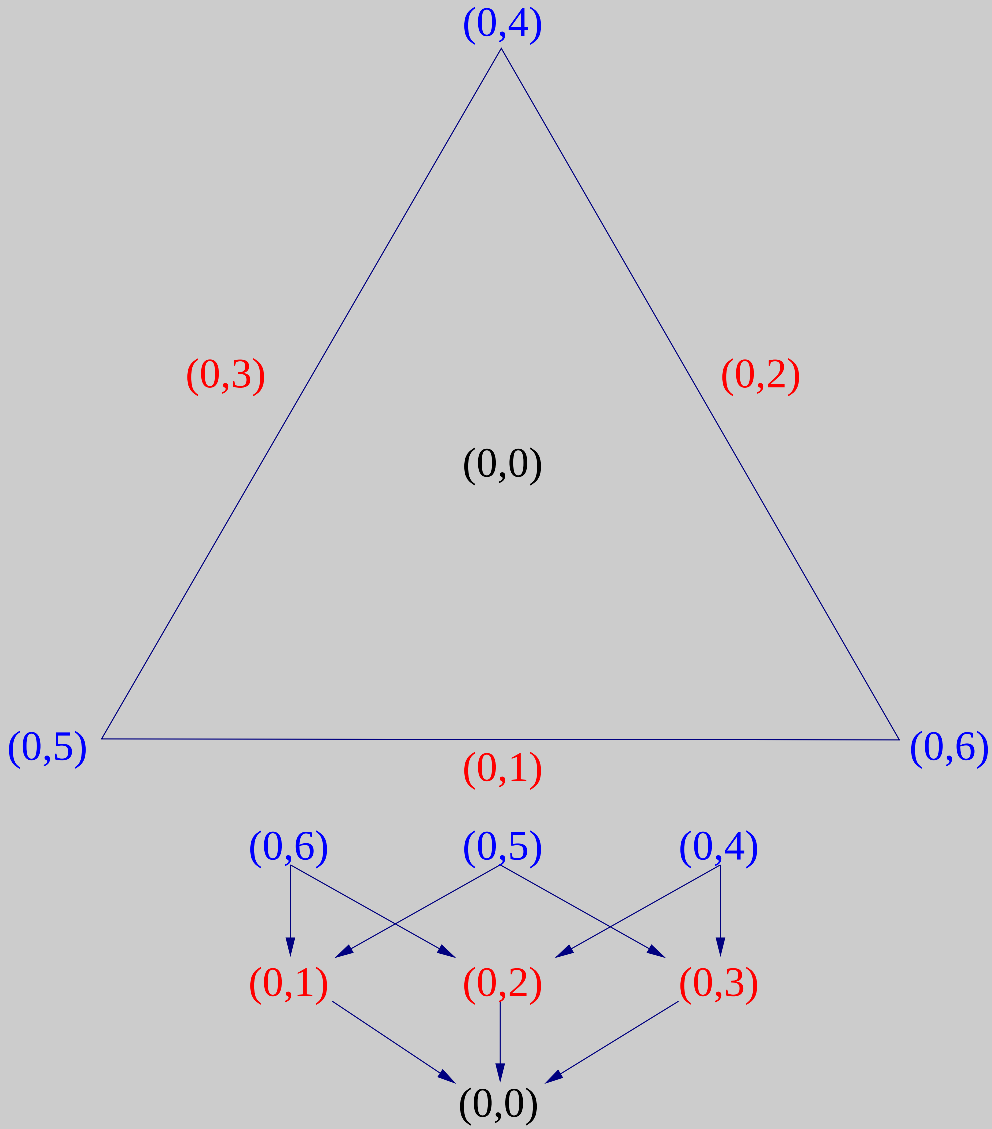
What is a Sieve?

A Sieve encodes topology

- Category with arrows denoting a *covering* relation
  - We say that *cap* elements cover *base* elements
  - Any set of elements is called a *chain*
- Model for set theory
- Hierarchical geometric data
  - Finite element meshes
  - Multipole octree
- Clean separation between topology and data organized by the topology
  - Con-fused in most packages, e.g. PETSc Vec

Simple Sieve

Topological elements are encoded as (process, local id)



Sieve Primitives

Cone: The set of cap elements covering a base element

$$cone(0,0) = \{(0,1), (0,2), (0,3)\}$$

Closure: The iterated cone

$$closure(0,0) = \{(0,1), (0,2), (0,3), (0,4), (0,5), (0,6)\}$$

Support: The set of base elements covered by a cap element

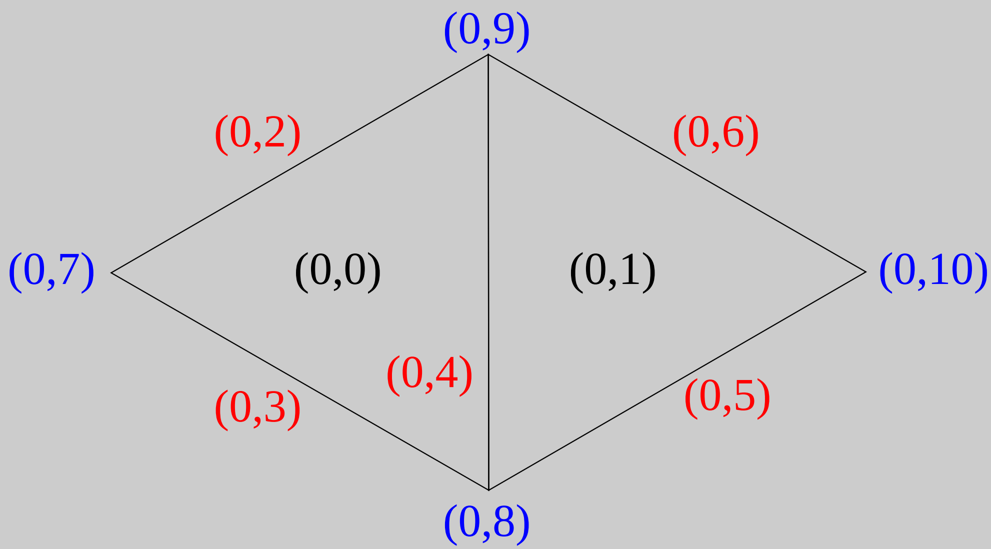
$$support(0,4) = \{(0,2), (0,3)\}$$

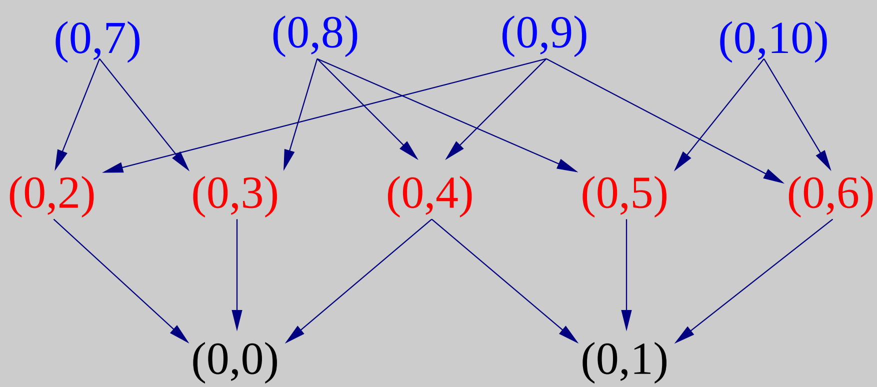
Star: The iterated support

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

Doublet Mesh

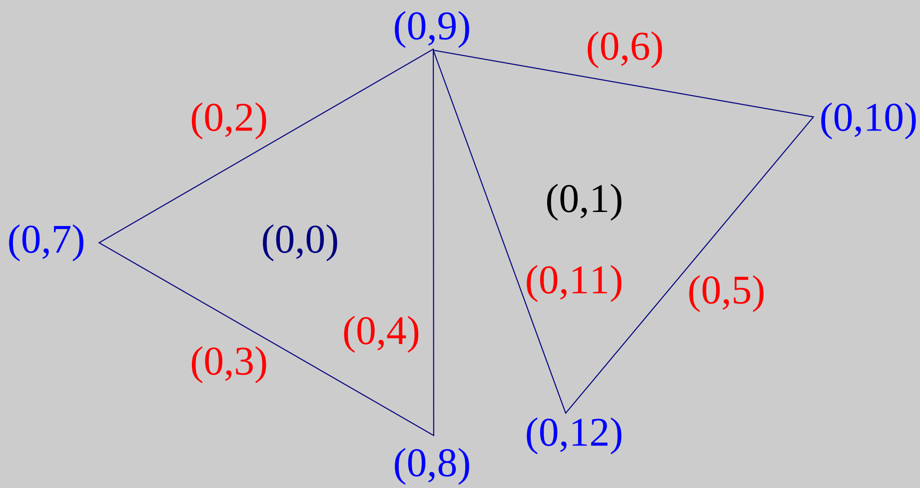
We can examine the meet and join using two adjacent elements

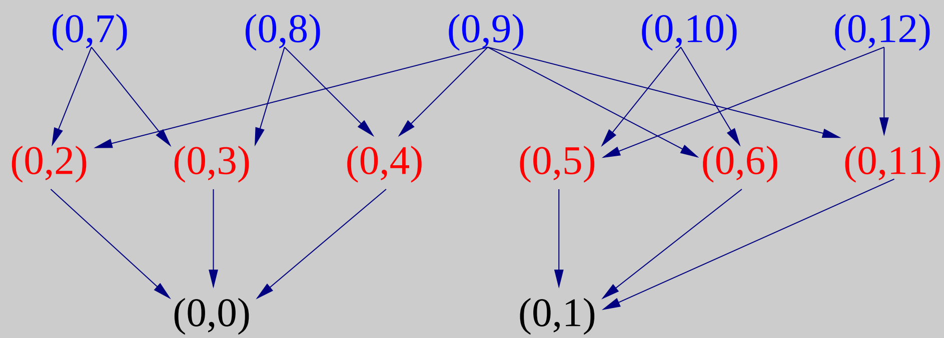




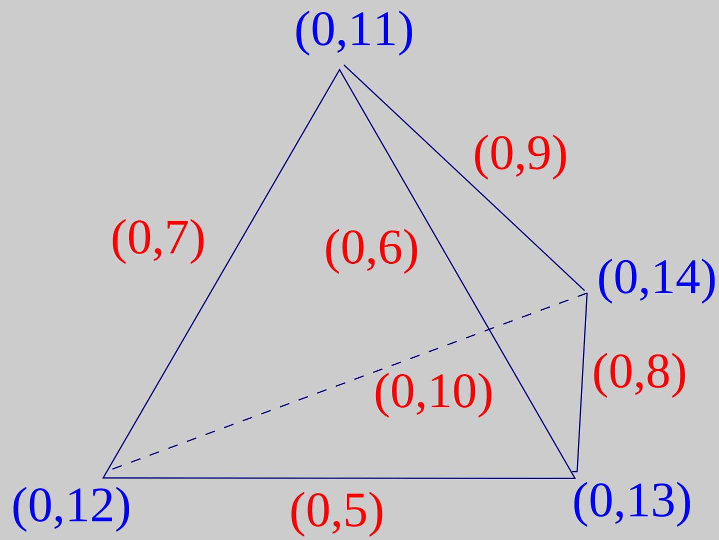
Doublet Mesh II

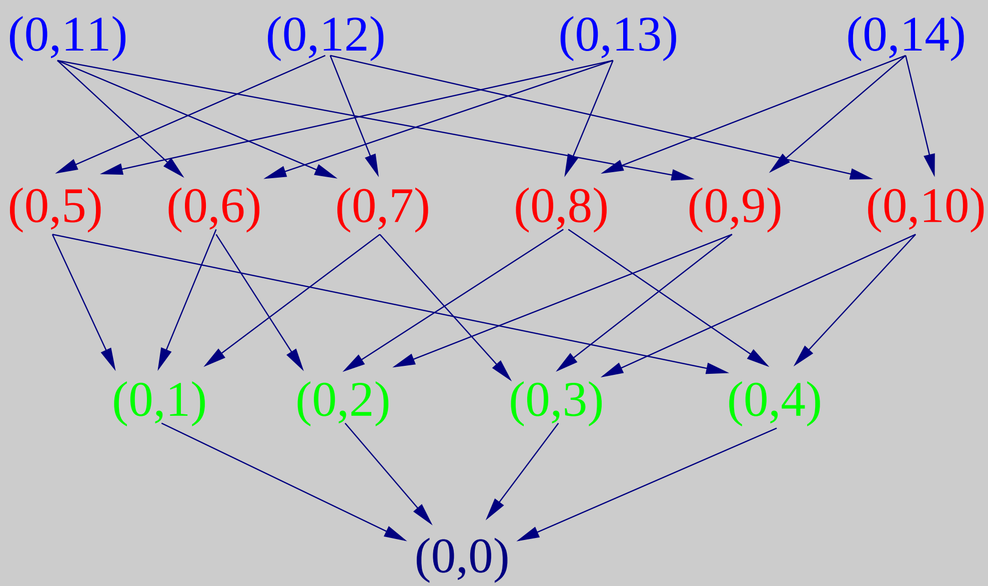
These elements provide a different lattice





Tetrahedron Mesh





Lattice Operations

Meet: The smallest set of elements whose star contains the given chain

- Can be seen as the intersection of the closures of the chain elements
- For the doublet mesh, $meet((0,0),(0,1)) = (0,4)$
- For the split doublet mesh, $meet((0,0),(0,1)) = (0,9)$

Join: The smallest set of elements whose closure contain the given chain

- Can be seen as the intersection of the supports of the chain elements
- For the doublet mesh, $join((0,0),(0,1)) = ((0,0),(0,1))$
- For the tetrahedron, $join((0,5),(0,7)) = (0,1)$
- However, also for the tetrahedron, $join((0,5),(0,9)) = (0,0)$

Cone Completion

In a distributed Sieve, parts of an elements's cone may lie on different processes. *Completion* constructs another local Sieve which contains the missing parts of each local cone.

- Dual operation of *support completion*
  - Uses the same communication routine
- Single parallel operation is sufficient for Sieve
- Enables many other parallel operations
  - Dual graph construction
  - Graph partitioning
  - Parallel and periodic meshing

The Sieved Array

Restriction

Restriction is the dual operation to covering

- Allows global fields to be manipulated locally
  - This is the heart of FEM
- Ties value storage to the topology (hierarchy)
- Can apply to any mesh subset (chain)
  - Single element
  - Mesh boundary
  - Local submesh
- Looks like indexing with elements

Sieved Arrays

- Represent values organized by the underlying topology
  - Solution fields
  - Mesh geometry
  - Boundary markers
  - Chemical species
- Allows natural operations of restriction and prolongation (assembly)
  - Many different storage policies may be used
- Allows user to work completely locally, letting the Array handle assembly
  - Very similar to PETSc strategy for parallelism
- Arrays are sections of a fibre bundle over the mesh
  - Transition between chains is a (nontrivial) map between vector spaces

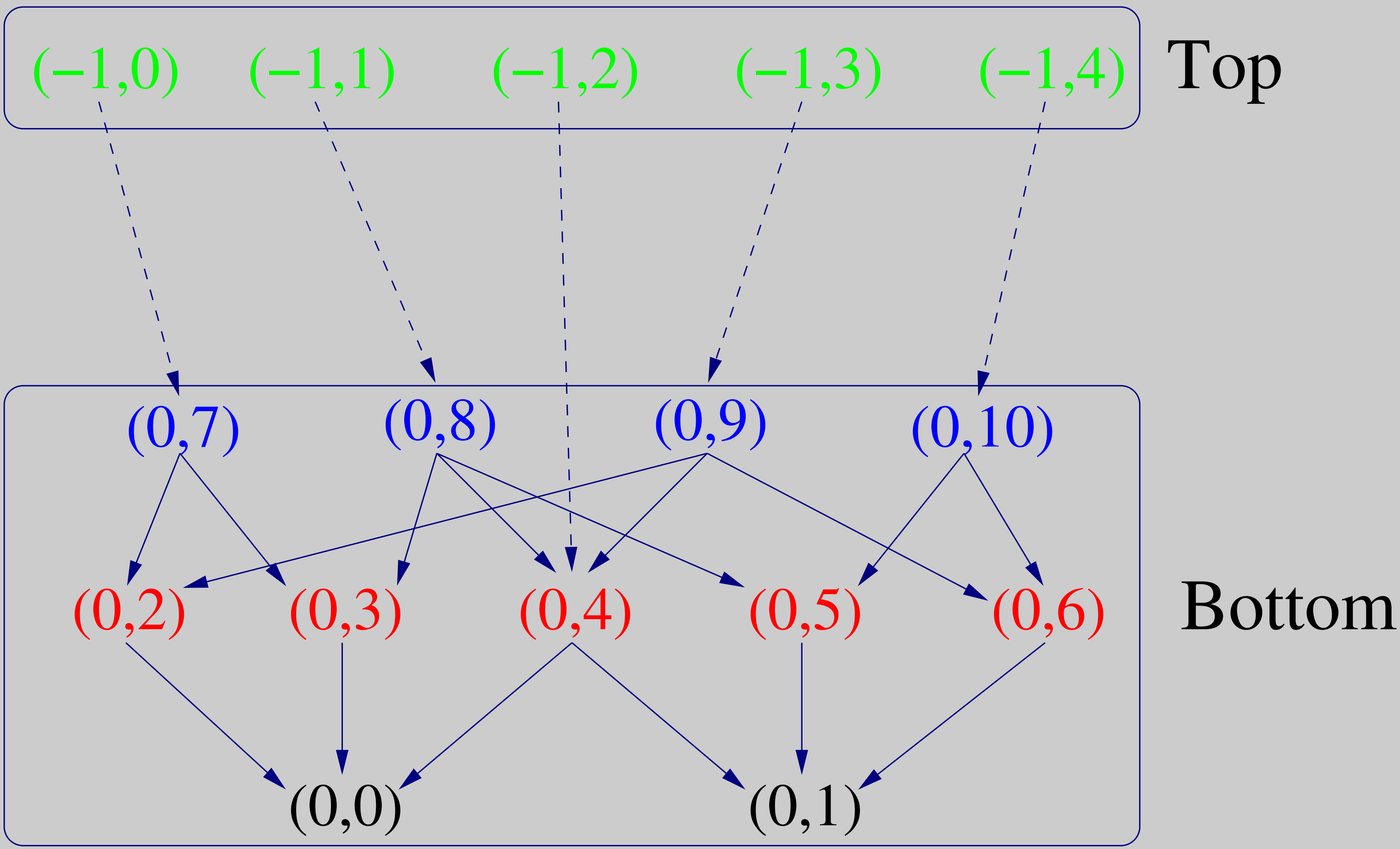
Sifting

Sifting is the operation of restricting an Array to a chain

- Nontrivial assembly and restriction policies
  - replacement/preservation
  - addition
  - coordinate transformation
  - orientation using the input chain
  - Nonconforming overlapping grids
- Decouples storage/restriction policy from continuum mathematics
  - Vectors are not Arrays
- Seems to tied to the storage to factor out

Stack

A *Stack* connects two Sieves with *vertical* arrows



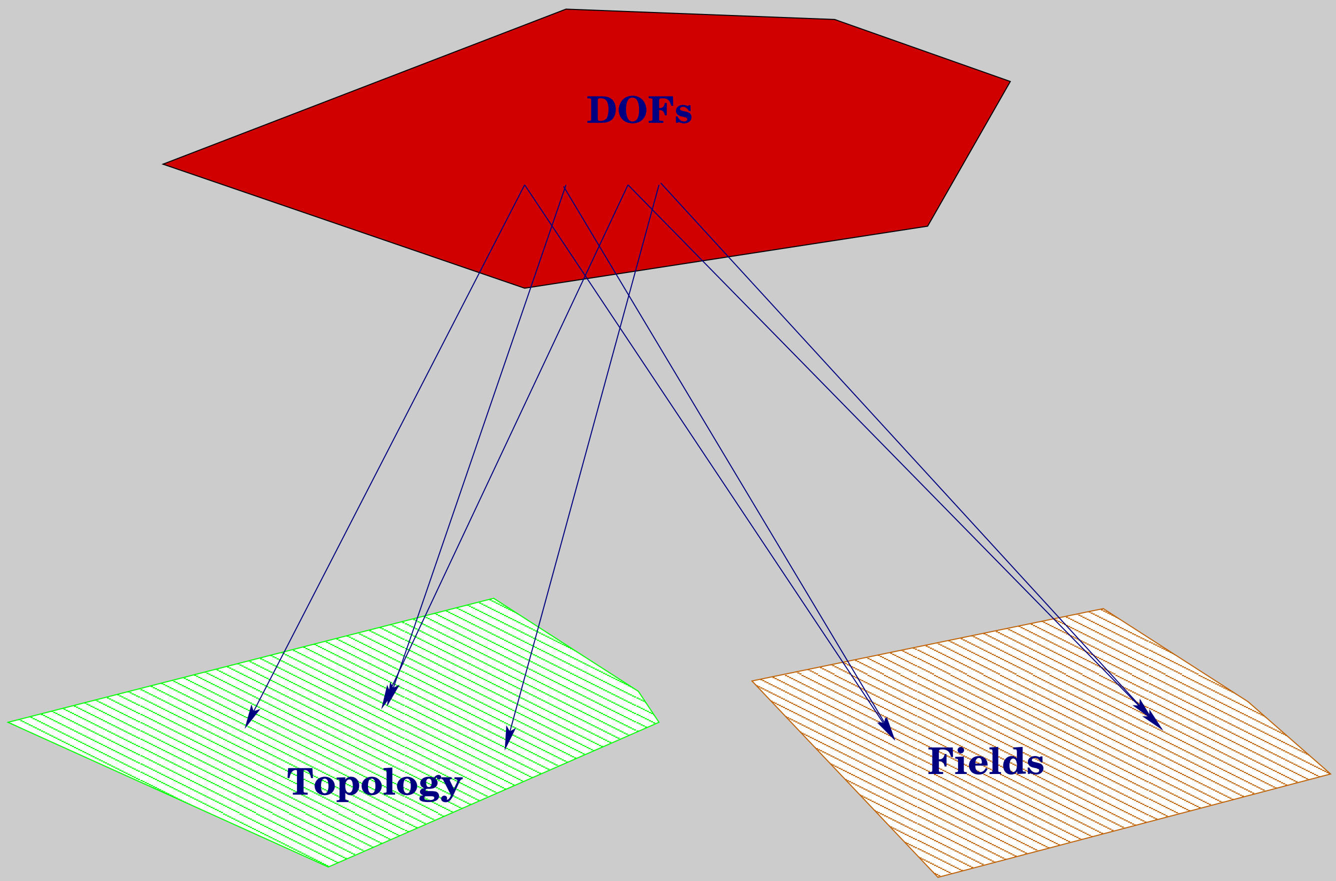
Degrees of Freedom

Stacks organize the degrees of freedom over a mesh

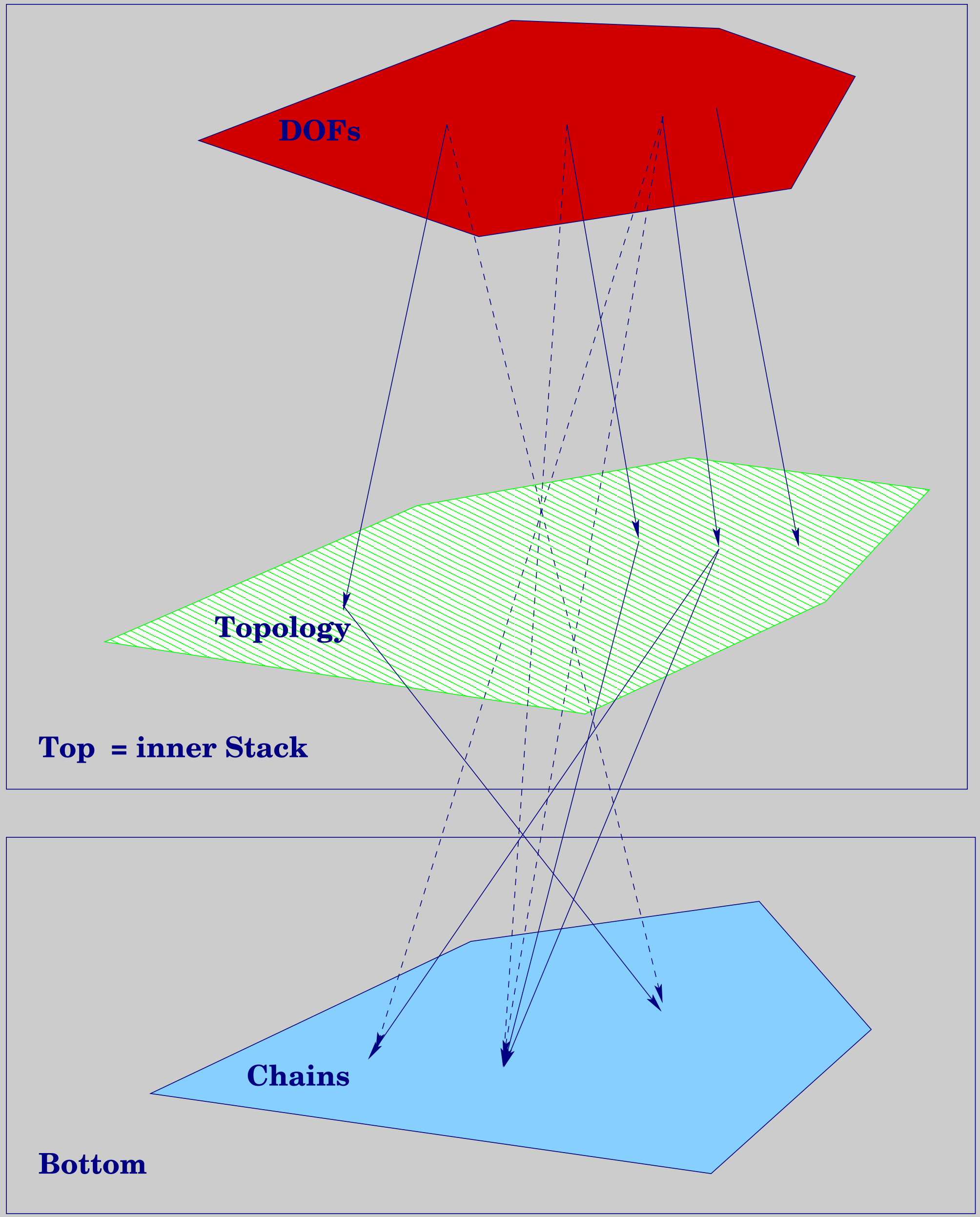
- The top (discrete) sieve contains the degrees of freedom
- The bottom sieve is the mesh topology
- Sieve operations now occur over vertical arrows
- Lattice operations will now have *pullback* and *pushforward* versions

Degrees of Freedom for Multiple Fields

Using a DOF and Field Stack with common top Sieve, we can extract the variables from a given field using the *meet* operation.



Trial Implementation



Examples

Dual Graph Creation

```
topology = mesh.getTopology()
# Loop over all edges
completion, footprint = topology.supportCompletion(supportFootprint)
for edge in topology.heightStratum(1):
  support = topology.support(edge)
  if len(support) == 2:
    dualTopology.addCone(support, edge)
  elif len(support) == 1 and completion.capContains(edge):
    cone = (support[0], completion.support(edge)[0])
    dualTopology.addCone(cone, edge)
dualMesh.setTopology(dualTopology)
```

Mesh Partitioning

```
def partitionDoublet(self, topology):
  if rank == 0:
    topology.addCone(topology.closure((0, 0)), (-1, 0))
    topology.addCone(topology.closure((0, 1)), (-1, 1))
  else:
    topology.addBasePoint((-1, rank))
```

Mesh Partitioning

```
def genericPartition(self, comm, topology):
  # Cone complete to move the partitions to the other processors
  completion, footprint = topology.coneCompletion(footprintTypeCone)
  # Merge in the completion
  topology.add(completion)
  # Cone complete again to build the local topology
  completion, footprint = topology.coneCompletion(footprintTypeCone)
  # Merge in the completion
  topology.add(completion)
  # Restrict to the local partition
  topology.restrictBase(topology.cone((-1, rank)))
  # Optional: Support complete to get the adjacency information
```

FEM Numbering

Start by creating the discretizations and a Stack

```
elements = [FIAT.Lagrange.Lagrange(FIAT.shapes.TRIANGLE, 2),
            FIAT.Lagrange.Lagrange(FIAT.shapes.TRIANGLE, 3)]
ranks = [1, 0]
dof = ALE.Sieve.Sieve()
numbering = ALE.Stack.Stack()
numbering.setTop(dof)
numbering.setBottom(topology)
```

FEM Numbering

```
def multipleFieldsStack(self, topology):
  completion, footprint = topology.supportCompletion(supportType)
  for p in topology.space():
    if completion.capContains(p):
      support = footprint.support([p]+list(completion.support(p)))
      if [0 for processTie in support if processTie[1] < rank]:
        continue
    indices = []
    for field in range(len(elements)):
      entityDof = len(dualBases[field].getNodeIDs(topology.depth(p))[0])
      tensorSize = entityDof*max(1, dim*ranks[field])
      var = [(-(rank+1), index+i) for i in range(tensorSize)]
      indices.extend(var); index += dof
      dof.addCone(var, (-1, field))
    numbering.addCone(var, p)
  completion, footprint = topology.coneCompletion(coneType)
```

FEM Assembly

```
elements = mesh.heightStratum(0)
elemU = u.restrict(elements)
# Loop over highest dimensional elements
for element in elements:
  # We want values over the element and all its coverings
  chain = mesh.closure(element)
  # Retrieve the field coefficients for this element
  coeffs = elemU.getValues([element])
  # Calculate the stiffness matrix and load vector
  K, f = self.integrate(coeffs, self.jacobian(element, mesh, space))
  # Place results in global storage
  elemF.setValues([chain], f)
  elemA.setValues([[chain], [chain]], K)
F = elemF.prolong([])
A = elemA.prolong([])
```

FEM Assembly

Notice that the prior code is independent of:

- dimension
- element type
- finite element
- sifting policy

Conclusions

Better mathematical abstractions bring concrete benefits:

- Vast reduction in complexity
  - Dimension independent code
  - Only a single communication routine to optimize
  - One relation handles all hierarchy
- Expansion of capabilities
  - Can handle hybrid meshes
  - Can hande complicated topologies (magnetization)
  - Can hande complicated structures (faults)