FEniCS and Sieve Tutorial

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 5, 2007
Workshop on Automating the Development of Scientific Computing Software
LSU, Baton Rouge, LA

Tutorial Goals

- Introduce FEniCS Automated Mathematical Modeling paradigm
- Enable students to develop new simulations with FEniCS
  - Demonstrate sample problems and typical operations
- Describe PETSc-Sieve project
  - High performance parallel infrastructure

Tutorial Goals

- Introduce FEniCS Automated Mathematical Modeling paradigm
- Enable students to develop new simulations with FEniCS
  - Demonstrate sample problems and typical operations
- Describe PETSc-Sieve project
  - High performance parallel infrastructure

Tutorial Goals

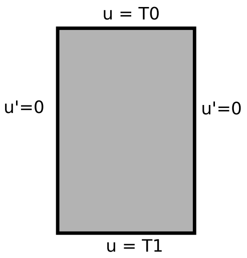
- Introduce FEniCS Automated Mathematical Modeling paradigm
- Enable students to develop new simulations with FEniCS
  - Demonstrate sample problems and typical operations
- Describe PETSc-Sieve project
  - High performance parallel infrastructure

Outline

FEM at a Glance
Strong Form

Find u on domain Ω, given f and BC

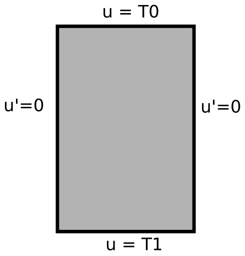
$$-\Delta u = f$$



FEM at a Glance
Weak Form

Find u on domain Ω, given f and BC,
such that for all v in the function space S

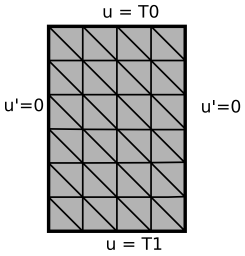
$a(u,v) = (f,v)$



FEM at a Glance
Discretization

Find u_h on a triangulization of domain Ω,
given f and BC,
such that for all v in the function space S

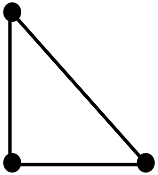
a(u_h,v) = (f,v)

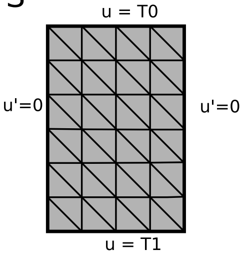


FEM at a Glance
Discretization

Find u_h on a triangulization of domain Ω,
given f and BC,
such that for all v_h
in the function space V ⊂ S

a(u_h,v_h) = (f,v_h)





FEM at a Glance
Discrete System

$$\Bigg[\quad A \quad\Bigg]\Bigg[\, x \,\Bigg] = \Bigg[\, b \,\Bigg]$$

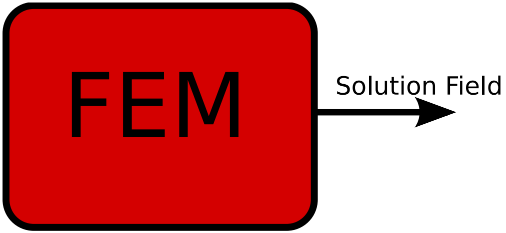
Outline

Outline

The FEniCS Project

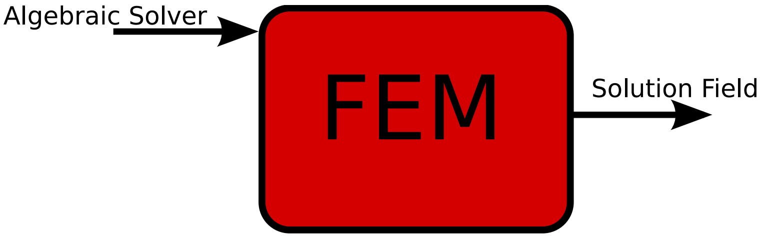
- Started in 2003 as a collaboration between
  - Chalmers
  - University of Chicago
- Now spans
  - KTH
  - University of Oslo and Simula Research
  - University of Chicago and Argonne National Laboratory
  - Cambridge University
  - TU Delft
- Focused on Automated Mathematical Modelling
- Allows researchers to easily and rapidly develop simulations

The FEniCS Project



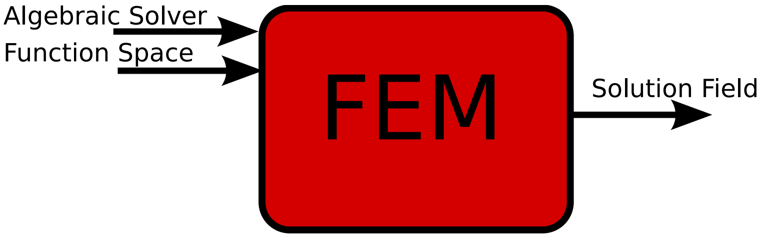
DOLFIN: The simulation engine which pulls all the pieces together.

The FEniCS Project



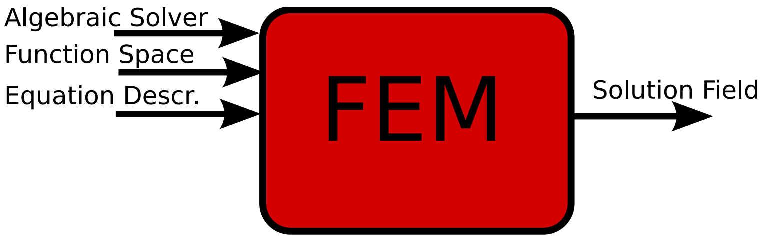
PETSc, Trilinos, MTL, uBlas, UMFPACK (separate projects outside FEniCS)

The FEniCS Project



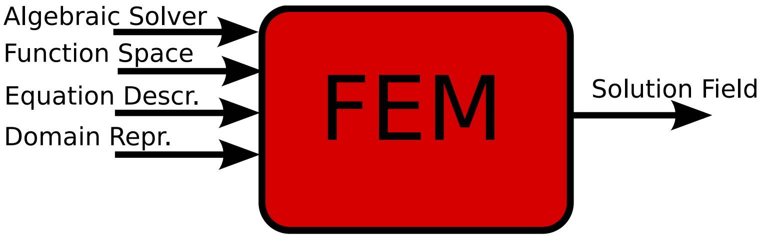
FIAT: Finite element Integrator And Tabulator
SyFi: SYmbolic FInite elements

The FEniCS Project



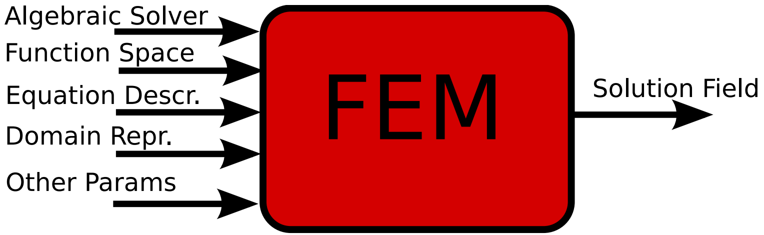
FFC: Fenics Form Compiler, or SyFi

The FEniCS Project



DOLFIN Mesh Library

The FEniCS Project

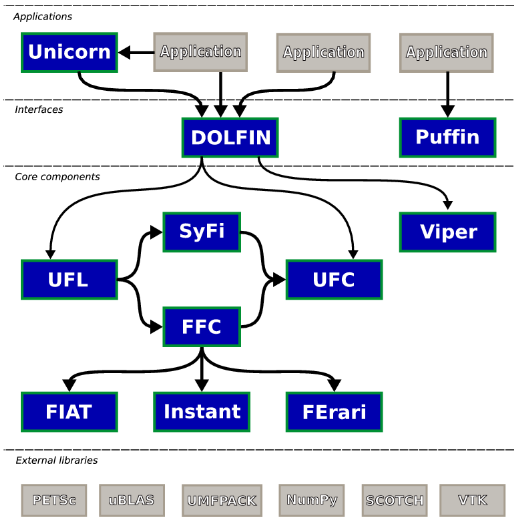


The FEniCS Project

Other projects

Project	Description
UFC	Links equation discretization to algebraic solver
Viper	Uses pyvtk to produce quick plots
Instant	JIT C compiler for inline functions in python
Puffin	Educational project
FErari	Optimizations for evaluation of variational forms
Sieve	Abstractions for parallel mesh and function representation

The FEniCS Project



Outline

What is PETSc?

A freely available and supported research code for the parallel solution of nonlinear algebraic equations

Free

- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users

Supported

- 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, Matlab, Julia, and Python

What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT6, BG/Q, NVIDIA Fermi, K Computer
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, iPad/iPhone, PCs running Linux or Windows
- PETSc History
  - Begun September 1991
  - Over 60,000 downloads since 1995 (version 2)
  - Currently 400 per month
- PETSc Funding and Support
  - Department of Energy
    - SciDAC, MICS Program, AMR Program, INL Reactor Program
  - National Science Foundation
    - CIG, CISE, Multidisciplinary Challenge Program

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media
- PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL
- PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code

Outline

Download and Install
Debian Packages

- UFC:
  `apt-get install ufc`
- FIAT:
  `apt-get install fiat`
- FFC:
  `apt-get install ffc`
- DOLFIN:
  `apt-get install dolfin`
- Viper:
  `apt-get install dolfin`

You also need
```
   deb http://www.fenics.org/debian/ unstable main
   deb-src http://www.fenics.org/debian/ unstable main
in your /etc/apt/source.list, and the key
   wget http://www.fenics.org/debian/pubring.gpg -O- | sudo apt-key add -
```

Download and Install

Source Tarballs

- UFC:
  `http://www.fenics.org/pub/software/ufc/v1.0/ufc-1.1.tar`
- FIAT:
  `http://www.fenics.org/pub/software/fiat/FIAT-0.3.4.tar.`
- FFC:
  `http://www.fenics.org/pub/software/ffc/v0.4/ffc-0.4.4.t`
- DOLFIN:
  `http://www.fenics.org/pub/software/dolfin/v0.7/dolfin-0`
- Viper:
  `http://www.fenics.org/pub/software/viper/v0.2/viper-0.2`

Download and Install
Mercurial Repositories

- UFC:
```
hg clone http://www.fenics.org/hg/ufc
python setup.py install
```
- FIAT:
```
hg clone http://www.fenics.org/hg/fiat
python setup.py install
```
- FFC:
```
hg clone http://www.fenics.org/hg/ffc
python setup.py install
```
- DOLFIN:
```
hg clone http://www.fenics.org/hg/dolfin
See http://www.fenics.org/wiki/DOLFIN
```
- Viper:
```
hg clone http://www.fenics.org/hg/viper
python setup.py install
```

Cloning PETSc

- The full development repository is open to the public
  - https://bitbucket.org/petsc/petsc/
- Why is this better?
  - You can clone to any release (or any specific ChangeSet)
  - You can easily rollback changes (or releases)
  - You can get fixes from us the same day
- All releases are just tags:
  - Source at tag v3.4.4

Automatic Downloads

- Starting in 2.2.1, some packages are automatically
  - Downloaded
  - Configured and Built (in `$PETSC_DIR/externalpackages`)
  - Installed with PETSc
- Currently works for
  - petsc4py
  - PETSc documentation utilities (Sowing, lgrind, c2html)
  - BLAS, LAPACK, BLACS, ScaLAPACK, PLAPACK
  - MPICH, MPE, OpenMPI
  - ParMetis, Chaco, Jostle, Party, Scotch, Zoltan
  - MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack, pARMS
  - BLOPEX, FFTW, SPRNG
  - Prometheus, HYPRE, ML, SPAI
  - Sundials
  - Triangle, TetGen
  - FIAT, FFC, Generator
  - Boost

Outline

Outline

Simple Example: Poisson

Poisson

$$-\Delta u = f \qquad \text{on} \qquad \Omega = [0,1] \times [0,1]$$

- Define our Form and compile (FIAT + FFC)
- Define our Simulation (DOLFIN)
  - Define our mesh
  - Assemble and solve
  - Post process (visualize, error, ...)

Simple Example: Poisson
Defining the form

```
element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)
g = Function(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
a = dot(grad(v), grad(u))*dx
L = v*f*dx + v*g*ds
```

see `ffc/src/demo/Poisson.form`, and compile with

```
$ ffc Poisson.form
```

Simple Example: Poisson

Writing the Simulation: Define our mesh

```
UnitSquare mesh(32, 32);
```

- Need to give boundary conditions
- Could use other meshing tools and convert to Dolfin xml format

Simple Example: Poisson

Writing the Simulation: Assemble and solve

```
// Create user defined functions
Source f(mesh); Flux g(mesh);
// Create boundary condition
Function          u0(mesh, 0.0);
DirichletBoundary boundary;
DirichletBC       bc(u0, mesh, boundary);
// Define PDE
PoissonBilinearForm a;
PoissonLinearForm   L(f, g);
LinearPDE           pde(a, L, mesh, bc);
// Solve PDE
Function u;
pde.solve(u);
```

Simple Example: Poisson

Writing the Simulation: Post process

```
// Plot solution
plot(u);
// Save solution to file
File file("poisson.pvd");
file « u;
```

Simple Example: Poisson

Now let's define our source term as:

$$f(x,y) = 500 * \exp\left(-\frac{(x-0.5)^2 + (y-0.5)^2}{0.02}\right)$$

```
class Source : public Function {
  Source(Mesh& mesh) : Function(mesh) {};
  real eval(const real* x) const {
    real dx = x[0] - 0.5;
    real dy = x[1] - 0.5;
    return 500.0*exp(-(dx*dx + dy*dy)/0.02);
  }
};
```

Simple Example: Poisson

Boundary conditions given by

$$\begin{aligned} u(x,y) &= 0 && \text{for} \quad x = 0 \\ du/dn(x,y) &= 25\sin(5\pi y) && \text{for} \quad x = 1 \\ du/dn(x,y) &= 0 && \text{otherwise} \end{aligned}$$

```
class DirichletBoundary : public SubDomain {
  bool inside(const real* x, bool on_boundary) const
    return x[0] < DOLFIN_EPS && on_boundary;}
};
class Flux : public Function {
  Flux(Mesh& mesh) : Function(mesh) {};
  real eval(const real* x) const {
    if (x[0] > DOLFIN_EPS)
      return 25.0*sin(5.0*DOLFIN_PI*x[1]);
    else return 0.0;}
};
```

Simple Example: Poisson

Include headers and your done[1]

```
#include <dolfin.h>
#include "Poisson.h"
using namespace dolfin;
```

[1] See `dolfin/src/demo/pde/poisson/cpp`

Simple Example: Poisson

Simulate!

Outline

Example: High Order Poisson
Poisson

This time use higher order Lagrangian elements

$$-\Delta u = f \qquad \text{on} \qquad \Omega = [0,1] \times [0,1]$$

- Define our Form and compile (FIAT + FFC)
- Define our Simulation (DOLFIN)
    - Define our mesh
    - Assemble and solve
    - Post process (visualize, error, ...)

Example: High Order Poisson

Defining the form

```
element = FiniteElement("Lagrange", "triangle", p)

v = TestFunction(element)
u = TrialFunction(element)
f = Function(element)
g = Function(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
a = dot(grad(v), grad(u))*dx
L = v*f*dx + v*g*ds
```

Compile with

```
$ ffc HOPoisson.form
```

Example: High Order Poisson

Use the same DOLFIN code.

Simulate!

Outline

Example: Discontinuous Galerkin Poisson

Poisson

$$-\Delta u = f \qquad \text{on} \qquad \Omega = [0,1] \times [0,1]$$

Using a discontinuous Galerkin formulation (interior penalty method).

- Define our Form and compile (FIAT + FFC)
- Define our Simulation (DOLFIN)
  - Define our mesh
  - Assemble and solve
  - Post process (visualize, error, ...)

Example: Discontinuous Galerkin Poisson
Defining the form

```
element = FiniteElement("Discontinuous Lagrange",
                        "triangle", 1)
...
n = FacetNormal("triangle")
h = MeshSize("triangle")
alpha = 4.0; gamma = 8.0
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), mult(u, n))*ds
  - dot(mult(v, n), grad(u))*ds + gamma/h*v*u*ds
```

see `ffc/src/demo/PoissonDG.form`, and compile with

```
$ ffc PoissonDG.form
```

Example: Discontinuous Galerkin Poisson

Writing the Simulation: Assemble and solve

```
// Create user defined functions
Source f(mesh); Flux g(mesh);
FacetNormal n(mesh);
AvgMeshSize h(mesh);
// Define PDE
PoissonBilinearForm a;
PoissonLinearForm   L(f, g);
LinearPDE           pde(a, L, mesh, bc);
// Solve PDE
Function u;
pde.solve(u);
```

Example: Discontinuous Galerkin Poisson

Simulate!

Outline

Example: L2 Error Check

L2 Error:

$$||u - u_h||_{L^2(\Omega)}$$

- Define our Form and compile (FIAT + FFC)
- Add to our Simulation (DOLFIN)
  - Post process (visualize, error, ...)

Example: L2 Error Check

Defining the form

```
  P0 = FiniteElement("Discontinuous Lagrange", "triang
  Element1 = FiniteElement("Lagrange", "triangle", 1)

  U = Function(Element1)
  u = Function(Element1)
  v = BasisFunction(P0)

  e = U - u

  L = v*dot(e,e)*dx

$ ffc L2Error.form
```

Example: L2 Error Check

Writing the Simulation: Post process

```
ExactSolution U_ex;
Vector tmp;
L2Error::LinearForm L2Error(U,u);
FEM::assemble(L2Error, tmp, mesh);
real error = sqrt(fabs(tmp.sum()));
```

Outline

Stokes Equations: Basic Fluids Modeling

Function Space Matters

Stokes Equation

- Taylor-Hood
- Crouzeix-Raviart
- Iterated Penalty

$$\begin{aligned} -\Delta \mathbf{u} + \nabla \mathbf{p} &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

Stokes Equations: Basic Fluids Modeling

Function Space Matters

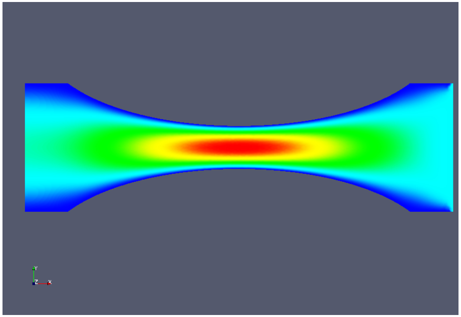
$$\frac{du}{dt} + u \cdot \nabla u = -\frac{\nabla \mathbf{p}}{\rho} + \nu \Delta \mathbf{u}$$

Stokes Equation
Taylor-Hood
Crouzeix-Raviart
Iterated Penalty

Navier-Stokes

- Stokes Solver
- Nonlinear Solver
- Time Stepping

Stokes Equations: Basic Fluids Modeling
Function Space Matters



Navier-Stokes
Stokes Solver
Nonlinear Solver
Time Stepping

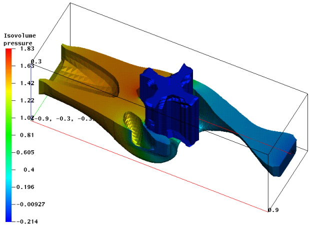
Stokes Equation
Taylor-Hood
Crouzeix-Raviart
Iterated Penalty

Non-Newtonian Flow

- Oldroyd-B
- Grade 2

Stokes Equations: Basic Fluids Modeling

Function Space Matters



Stokes Equation
Taylor-Hood
Crouzeix-Raviart
Iterated Penalty

Navier-Stokes
Stokes Solver
Nonlinear Solver
Time Stepping

Non-Newtonian
Odroyd-B
Grade 2
...

Fluid Solid Interfaces

- Free Boundary Problems
- Couple to legacy Codes

Outline

Stokes Mixed Methods
Stokes: Mixed Method Formulation

Let $V = H^1(\Omega)^n$ and $\Pi = \{q \in L^2(\Omega) : \int_\Omega q dx = 0\}$. Given $F \in V'$, find functions $\mathbf{u} \in V$ and $p \in \Pi$ such that

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = F(\mathbf{v}) \quad & \forall \mathbf{v} \in V \\ b(\mathbf{u}, q) = 0 \quad & \forall q \in \Pi \end{aligned}$$

Where,

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &:= \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx, \\ b(\mathbf{v}, q) &:= \int_\Omega (\nabla \cdot \mathbf{v}) q dx \end{aligned}$$

Stokes Mixed Method
Defining the form

```
P2 = VectorElement("Lagrange", "triangle", 2)
P1 = FiniteElement("Lagrange", "triangle", 1)
TH = P2 + P1

(v, q) = TestFunctions(TH)
(u, p) = TrialFunctions(TH)

f = Function(P2)

a = (dot(grad(v), grad(u)) - div(v)*p + q*div(u))*dx
L = dot(v, f)*dx
```

see
`dolfin/src/demo/pde/stokes/taylor-hood/cpp/Stokes.form`
and compile with

```
$ ffc Stokes.form
```

Stokes Mixed Method
Define our mesh

Use a predefined mesh, can be made with Triangle, Gmsh, ... and converted to DOLFIN mesh form with dolfin-convert
Use a MeshFunction to mark up different dof on boundary

```
// Read mesh and sub domain markers
Mesh mesh("dolfin-2.xml.gz");
MeshFunction<unsigned int> sub_domains(mesh,
                                        "subdomains.xml.gz");
```

Stokes Mixed Method
New Boundary Conditions

```
// Create functions for boundary conditions
Noslip noslip(mesh); Inflow inflow(mesh);
Function zero(mesh, 0.0);

// Define sub systems for boundary conditions
SubSystem velocity(0);
SubSystem pressure(1);

// BC's per field
DirichletBC bc0(noslip, sub_domains, 0, velocity);
DirichletBC bc1(inflow, sub_domains, 1, velocity);
DirichletBC bc2(zero, sub_domains, 2, pressure);
Array <BoundaryCondition*> bcs(&bc0, &bc1, &bc2);
```

Stokes Mixed Method
Assemble and solve

```
// Set up PDE
Function f(mesh, 0.0);
StokesBilinearForm a;
StokesLinearForm L(f);
LinearPDE pde(a, L, mesh, bcs);

// Solve PDE
Function u;
Function p;
pde.set("PDE linear solver", "direct");
pde.solve(u, p);
```

Stokes Mixed Method

Writing the Simulation: Post process

```
// Plot solution
plot(u);
plot(p);
// Save solution to file
File file("velocity.pvd");
file « u;
File file("pressure.pvd");
file « p;
```

Stokes Mixed Method

```
// Functions for boundary condition for velocity
class Noslip : public Function  {
public:
  Noslip(Mesh& mesh) : Function(mesh) {}
  void eval(real* values, const real* x) const {
    values[0] = 0.0;
    values[1] = 0.0;
  }
};
class Inflow : public Function  {
public:
  Inflow(Mesh& mesh) : Function(mesh) {}
  void eval(real* values, const real* x) const {
    values[0] = -1.0;
    values[1] = 0.0; }
};
```

Stokes Mixed Method

Simulate!

Outline

Iterated Penalty

Stokes: Iterated Penalty Formulation

Let $r \in \mathbb{R}$ and $\rho > 0$ define u^n and $p = w^n$ by

$$\begin{aligned} a(\mathbf{u^n}, \mathbf{v}) + r(\nabla \cdot \mathbf{u^n}, \nabla \cdot \mathbf{v}) &= F(\mathbf{v}) - (\nabla \cdot \mathbf{v}, \nabla \cdot \mathbf{w^n}) \\ \mathbf{w}^{n+1} &= \mathbf{w}^n + \rho \mathbf{u}^n \end{aligned}$$

Stokes IP Method
Defining the form

```
Element = FiniteElement("Vector Lagrange", "triangle",

U = TrialFunction(Element)
v = TestFunction(Element)
f = Function(Element)
w = Function(Element)
c = Constant()

a = (dot(grad(v), grad(U)) - c * div(U) * (div(v)))*dx
L = dot(v, f) * dx + dot(div(v),div(w))*dx
```

```
$ ffc Stokes.form
```

Stokes IP Method
Assemble and solve

Setup is relatively the same.

```
Function f(mesh, 0.0), w, u;
real rho, r, div_u_error;
Stokes::BilinearForm a(rho);
rho = r = 1.0e3;
w.init(mesh, a.trial());
```

Stokes IP Method
Assemble and solve

But we iterate our solution based on L2Error.

```
for(int j; j<MAX_ITERS; j++)
{
  Stokes::LinearForm L(f,w);
  PDE pde(a, L, mesh, bcs);
  // Compute solution
  pde.solve(U);
  Vector tmp = w.vector() + r * (U.vector());
  w = Function(tmp);
  L2div::LinearForm div_u(U);
  FEM::assemble(div_u, tmp, mesh);
  div_u_error = sqrt(fabs(tmp.sum()));
  if (div_u_error < 5.0e-7) break;
  }
```

Stokes IP Method

Simulate!

Questions

Fenics Webpage:
http://www.fenics.org/

Join the mailing lists!

Outline

Outline

Linear Algebra Abstractions

- Need clear interfaces to ALL levels in the conceptual hierarchy
- Abstractions allow reuse of iterative solvers (Krylov methods)
  - `Vec` and `Mat` objects
  - `KSP` uses only the action of `Mat` on `Vec`, `MatMult()`
- PETSc provides a range of data types
  - `MPIAIJ`, `MPIAIJPERM`, `SuperLU`, . . .
  - Arbitrary user code accomodated using `MATSHELL` objects

Solver Choice

- Can choose solver at runtime
  - `-ksp_type bicgstab`
- Can customize solver
  - `-ksp_gmres_restart 500`
  - Inapplicable options are ignored (same with API calls)
- Monitoring
  - `-ksp_monitor -ksp_view`

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

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

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

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

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

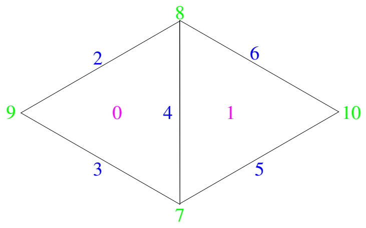
Unstructured Interface (after)

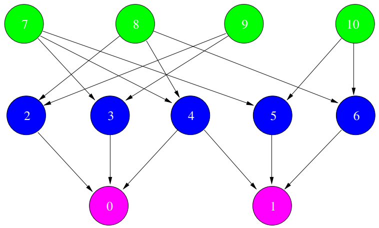
- NO explicit references to element type
  - A point may be any mesh element
  - getCone(point): adjacent (d-1)-elements
  - getSupport(point): adjacent (d+1)-elements
- Transitive closure
  - closure(cell): The computational unit for FEM
- Algorithms independent of mesh
  - dimension
  - shape (even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

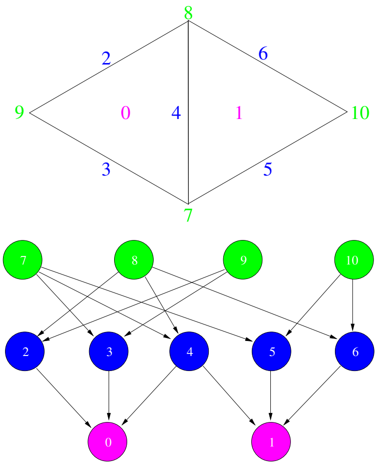
Doublet Mesh





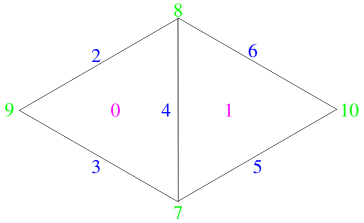
- Incidence/covering arrows
- *cone*(0) = {2, 3, 4}
- *support*(7) = {2, 3}

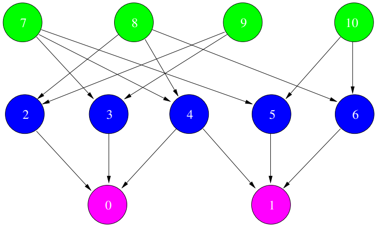
Doublet Mesh



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

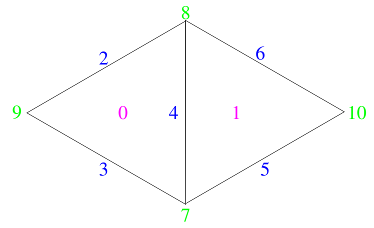
Doublet Mesh

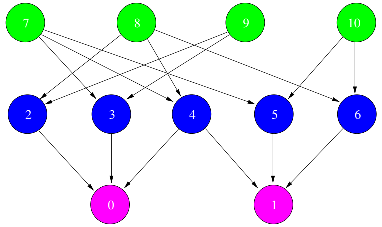




- Incidence/covering arrows
- *cone*$(0) = \{2, 3, 4\}$
- *support*$(7) = \{2, 3\}$

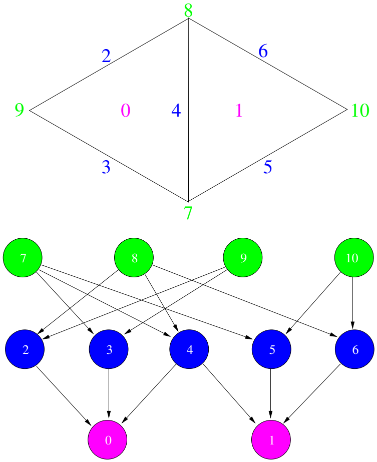
Doublet Mesh





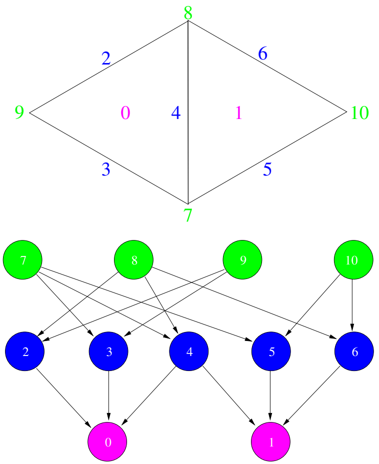
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



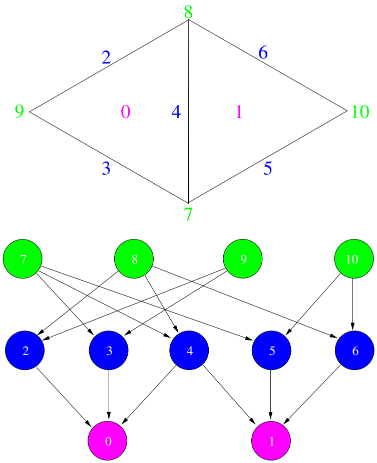
- Incidence/covering arrows
- $closure(0) = \{0, 2, 3, 4, 7, 8, 9\}$
- $star(7) = \{7, 2, 3, 0\}$

Doublet Mesh



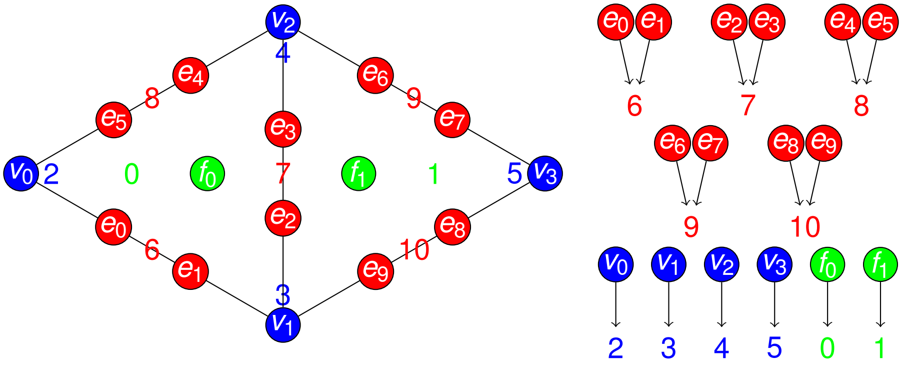
- Incidence/covering arrows
- *meet*$(0, 1) = \{4\}$
- *join*$(8, 9) = \{4\}$

Doublet Mesh



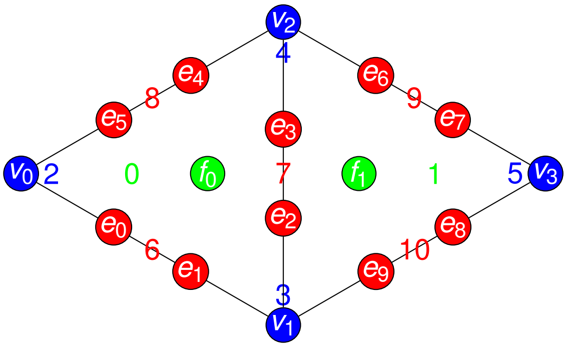
- Incidence/covering arrows
- $\mathit{meet}(0, 1) = \{4\}$
- $\mathit{join}(8, 9) = \{4\}$

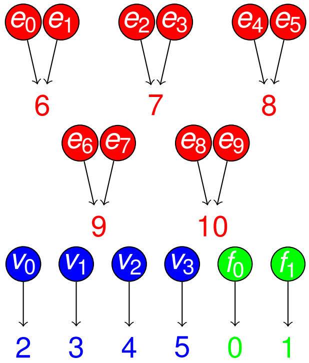
Doublet Section



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

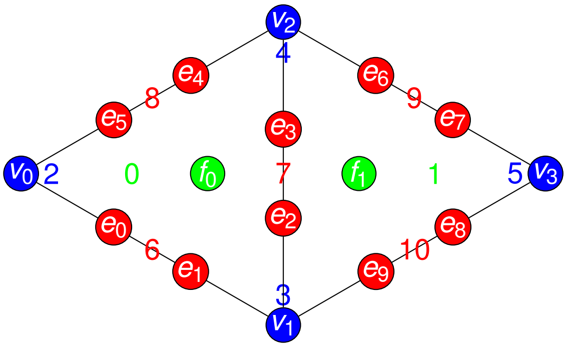
Doublet Section

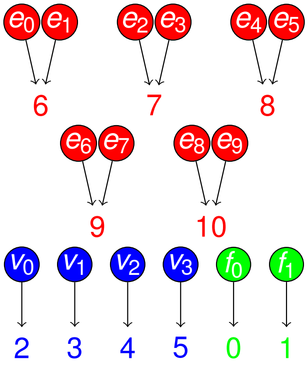




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

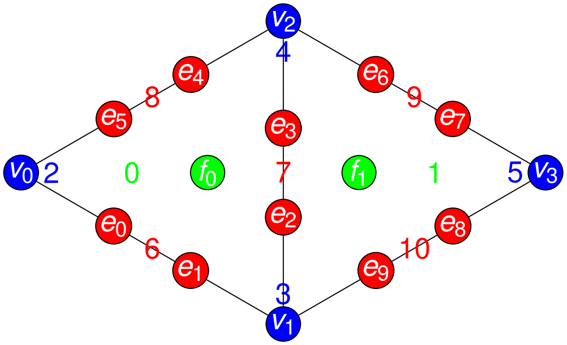
Doublet Section

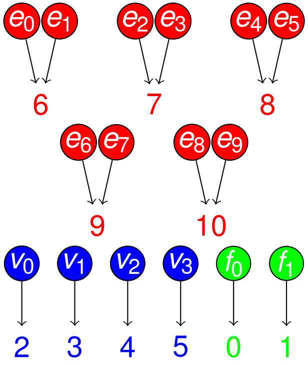




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

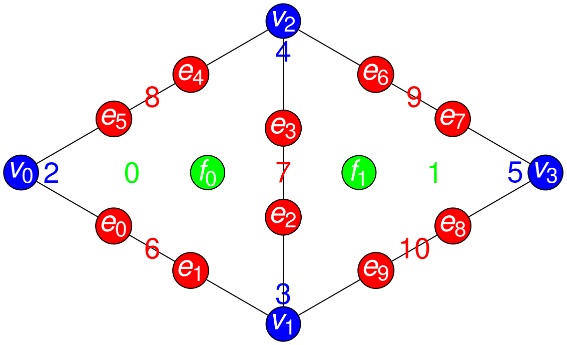
Doublet Section

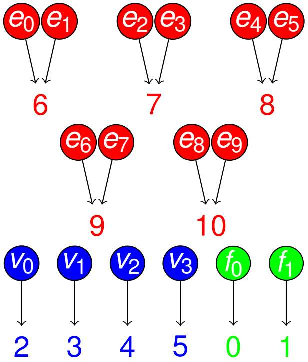




- **Section** interface
  - $restrict(0) = \{f_0\}$
  - $restrict(2) = \{v_0\}$
  - $restrict(6) = \{e_0, e_1\}$

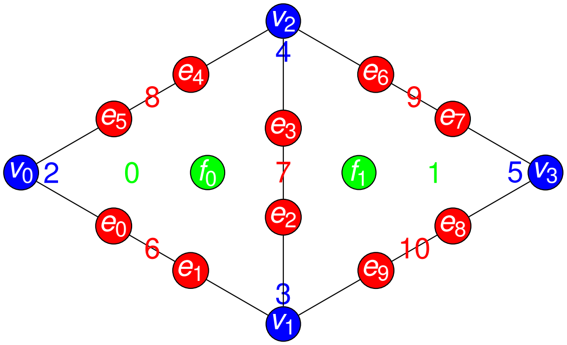
Doublet Section

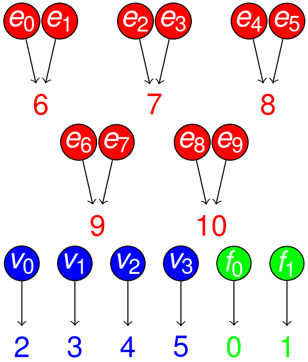




- Topological traversals: follow connectivity
  - *restrictClosure*$(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - *restrictStar*$(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

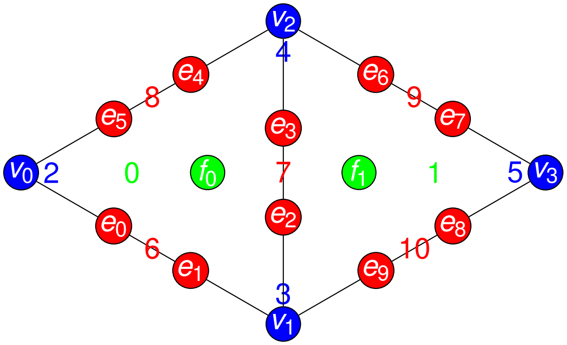
Doublet Section

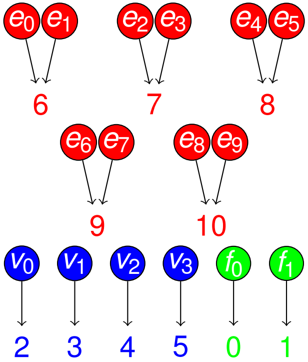




- Topological traversals: follow connectivity
  - *restrictClosure*$(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - *restrictStar*$(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

Doublet Section

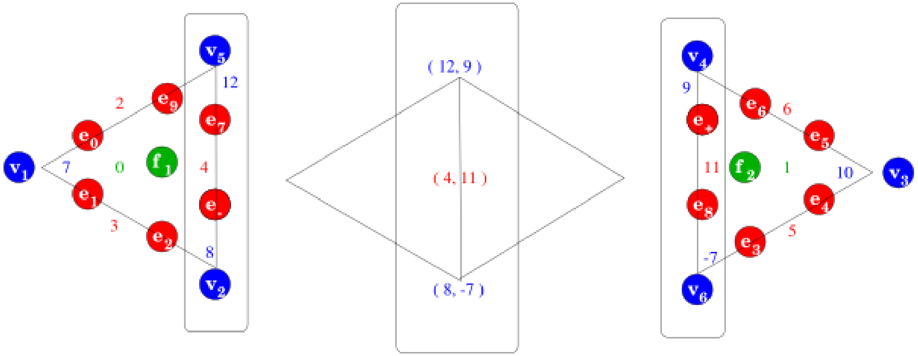




- Topological traversals: follow connectivity
  - $\mathit{restrictClosure}(0) = \{f_0\, e_0\, e_1\, e_2\, e_3\, e_4\, e_5\, v_0\, v_1\, v_2\}$
  - $\mathit{restrictStar}(7) = \{v_0\, e_0\, e_1\, e_4\, e_5\, f_0\}$

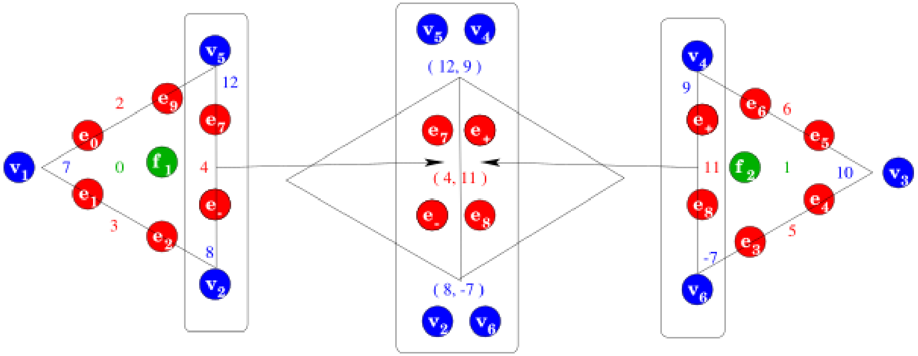
Outline

Restriction



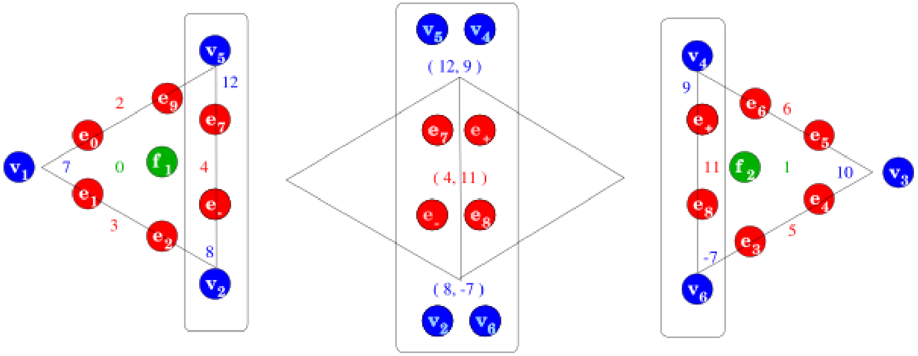
- Localization
  - Restrict to patches (here an edge closure)
  - Compute locally

Delta



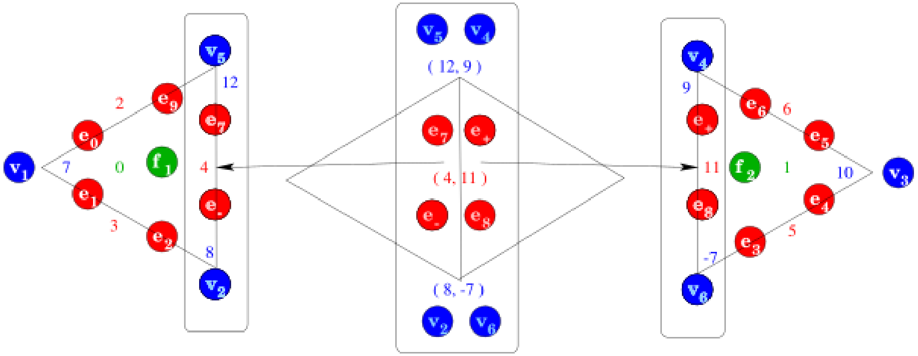
- `Delta`
  - Restrict further to the overlap
  - `Overlap` now carries twice the data

Fusion



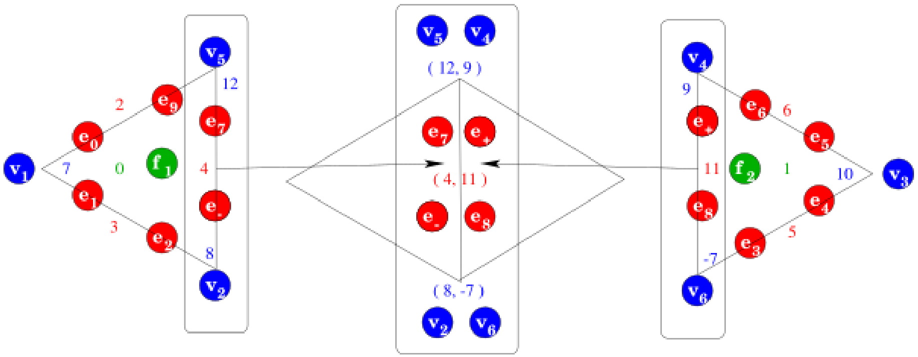
- Merge/reconcile data on the overlap
  - Addition (FEM)
  - Replacement (FD)
  - Coordinate transform (Sphere)
  - Linear transform (MG)

Update



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

Completion



- A ubiquitous <u>parallel</u> 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!

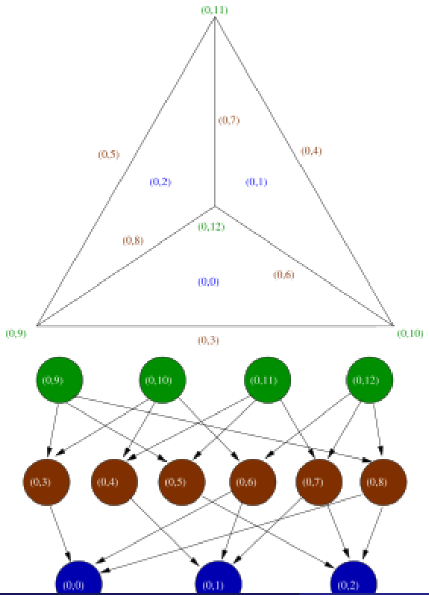
Mesh Distribution

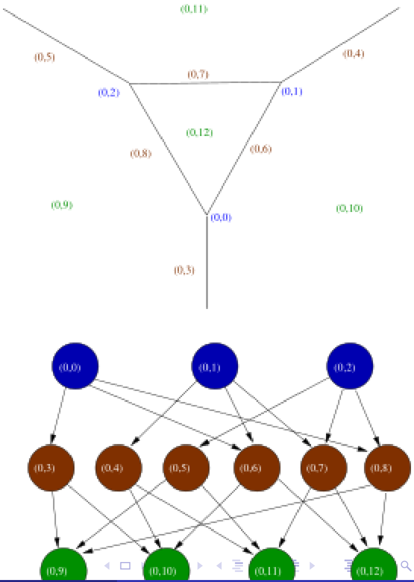
Distributing a mesh means

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

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

The Mesh Dual



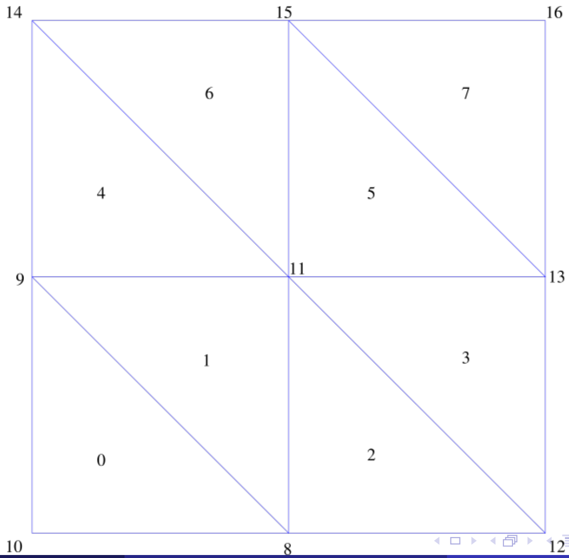


Mesh Partition

- 3rd party packages construct a vertex partition
- For FEM, partition dual graph vertices
- For FVM, construct hyperpgraph dual with faces as vertices
- Assign `closure(v)` and `star(v)` to same partition

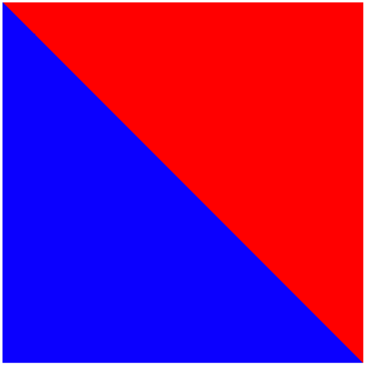
2D Example

A simple triangular mesh



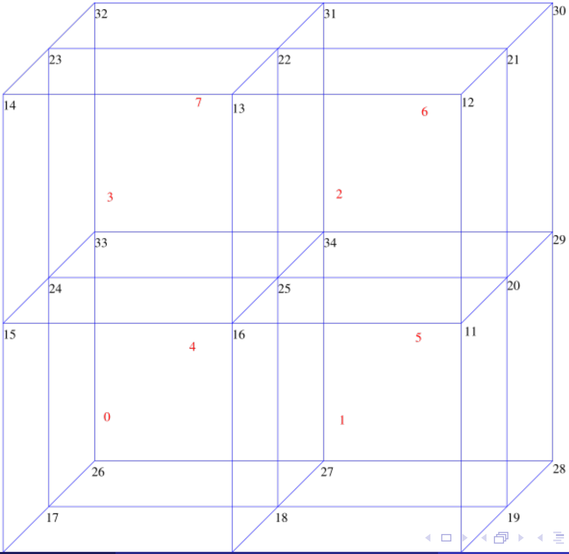
2D Example

Distributed Mesh



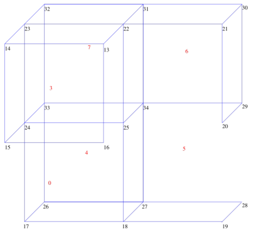
3D Example

A simple hexahedral mesh

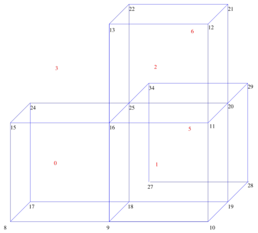


3D Example

Distributed Mesh



Process 0



Process 1

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

Dual Basis Application

We would like the action of a dual basis vector (functional)

$$< \mathcal{N}_i, f >= \int_{\mathrm{ref}} N_i(x) f(x) dV$$

- Projection onto $\mathcal{P}$
- Code is generated from FIAT specification
  - Python code generation package inside PETSc
- Common interface for all elements

Outline

What Is Optimal?

I will define *optimal* as an $\mathcal{O}(N)$ solution algorithm

These are generally hierarchical, so we need

- hierarchy generation
- assembly on subdomains
- restriction and prolongation

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Payoff

Why should I care?

1. Current algorithms do not efficiently utilize modern machines
2. Processor flops are increasing much faster than bandwidth
3. Multicore processors are the future
4. Optimal multilevel solvers are necessary

Claim: Hierarchical operations can be handled by a single interface

Why Optimal Algorithms?

- The more powerful the computer,
  the greater the importance of optimality
- Example:
  - Suppose Alg_1 solves a problem in time CN^2, N is the input size
  - Suppose Alg_2 solves the same problem in time CN
  - Suppose Alg_1 and Alg_2 are able to use 10,000 processors
- In constant time compared to serial,
  - Alg1 can run a problem 100X larger
  - Alg2 can run a problem 10,000X larger
- Alternatively, filling the machine's memory,
  - Alg1 requires 100X time
  - Alg2 runs in constant time

Multigrid

Multigrid is *optimal* in that is does $\mathcal{O}(N)$ work for $||r|| < \epsilon$

- Brandt, Briggs, Wan & Chan & Smith
- Constant work per level
  - Sufficiently strong solver
  - Need a constant factor decrease in the residual
- Constant factor decrease in dof
  - Log number of levels
- Sufficiently good interpolation
  - Preserves low modes
  - Cannot dump too much energy into high modes

Linear Convergence of the Poisson Problem

Convergence to $||r|| < 10^{-9}||b||$ using GMRES(30)/ILU

Elements	Iterations
128	10
256	17
512	24
1024	34
2048	67
4096	116
8192	167
16384	329
32768	558
65536	920
131072	1730

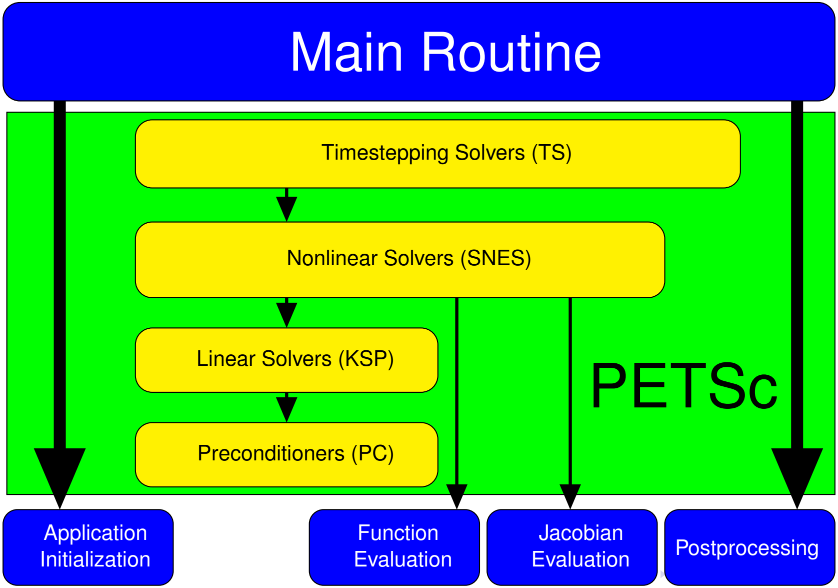
Linear Convergence of the Poisson Problem

Convergence to $||r|| < 10^{-9}||b||$ using GMRES(30)/MG

Elements	Iterations
128	5
256	7
512	6
1024	7
2048	6
4096	7
8192	6
16384	7
32768	6
65536	7
131072	6

Outline

Flow Control for a PETSc Application



SNES Paradigm

The SNES interface is based upon callback functions

- FormFunction(), set by SNESSetFunction()
- FormJacobian(), set by SNESSetJacobian()

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

- Solver calls the **user's** function
- User function gets application state through the `ctx` variable
  - PETSc never sees application data

Higher Level Abstractions

The PETSc `DA` class is a topology and discretization interface.

- Structured grid interface
  - Fixed simple topology
- Supports stencils, communication, reordering
  - Limited idea of operators
- Nice for simple finite differences

The PETSc `Mesh` class is a topology interface.

- Unstructured grid interface
  - Arbitrary topology and element shape
- Supports partitioning, distribution, and global orders

Higher Level Abstractions

The PETSc `DM` class is a hierarchy interface.

- Supports multigrid
  - `PCMG` combines it with a multigrid preconditioner
- Abstracts the logic of multilevel methods

The `PetscSection` class is a helper class for data layout.

- Functions over unstructured grids
  - Arbitrary layout of degrees of freedom
- Enables distribution and assembly

A DMDA is more than a Mesh

A DMDA contains topology, geometry, and (sometimes) an implicit Q1 discretization.

It is used as a template to create

- Vectors (functions)
- Matrices (linear operators)

Multigrid with DM

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

Creating a DMDA

DMDACreate2d(comm, bdX, bdY, type, M, N, m, n, dof, s, lm[], ln[], DMDA *da)

bd: Specifies boundary behavior

- DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, or DM_BOUNDARY_PERIODIC

type: Specifies stencil

- DMDA_STENCIL_BOX or DMDA_STENCIL_STAR

M/N: Number of grid points in x/y-direction

m/n: Number of processes in x/y-direction

dof: Degrees of freedom per node

s: The stencil width

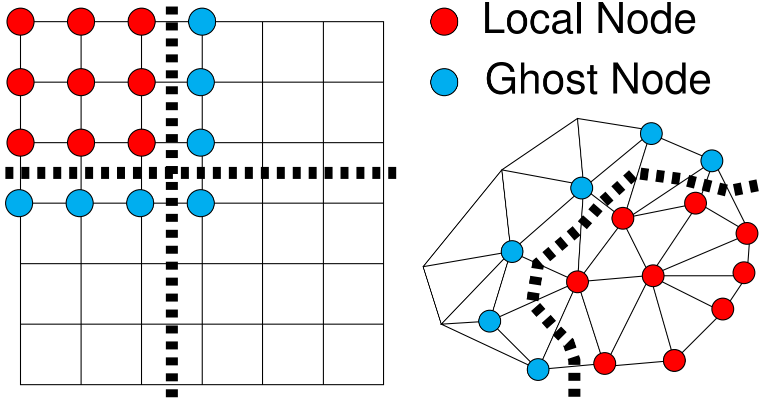
lm/n: Alternative array of local sizes

- Use NULL for the default

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 Global Numberings

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

DMDA Global vs. Local Numbering

- **Global**: Each vertex has a unique id belongs on a unique process
- **Local**: Numbering includes vertices from neighboring processes
  - These are called ghost vertices

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

DMDA Vectors

- 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!

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

$$\Delta u + \lambda e^u = 0$$

```
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)

```
(* ljac )(DMDALocalInfo *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

```
DMDASNESSetJacobianLocal(dm, ljac, &ctx)
```

Updating Ghosts

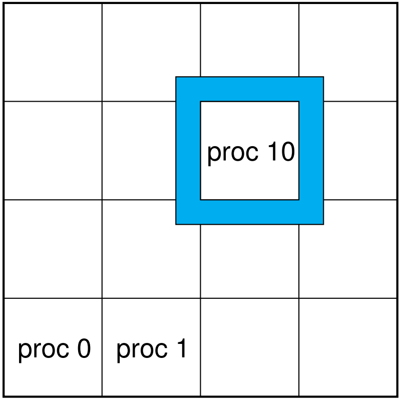
Two-step process enables overlapping computation and communication

- DMGlobalToLocalBegin(da, gvec, mode, lvec)
    - `gvec` provides the data
    - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
    - `lvec` holds the local and ghost values
- DMGlobalToLocalEnd(da, gvec, mode, lvec)
    - Finishes the communication

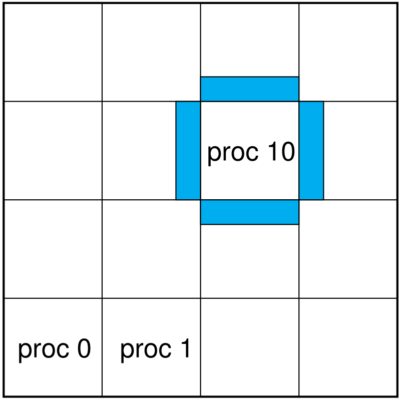
The process can be reversed with DALocalToGlobalBegin/End().

DMDA Stencils

Both the box stencil and star stencil are available.



Box Stencil



Star Stencil

Setting Values on Regular Grids

PETSc provides

```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n, MatStencil idxn[],
                    PetscScalar values[], InsertMode mode)
```

- Each row or column is actually a **MatStencil**
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are *vertices*
- The values are the same logically dense block in row/col

DM Integration with SNES

- 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()`

The Bratu Problem

$$\Delta u + \lambda e^{u} = f \quad \text{in} \quad \Omega \tag{1}$$

$$u = g \quad \text{on} \quad \partial\Omega \tag{2}$$

- Nonlinearly perturbed Poisson
- Can be treated as a nonlinear eigenvalue problem
- Has two solution branches until $\lambda \cong 6.28$

A 2D Problem

Problem has:

- 1,640,961 unknowns (on the fine level)
- 8,199,681 nonzeros

	Options	Explanation
`./ex5`	`-da_grid_x 21 -da_grid_y 21`	Original grid is 21x21
	`-ksp_rtol 1.0e-9`	Solver tolerance
	`-da_refine 6`	6 levels of refinement
	`-pc_type mg`	4 levels of multigrid
	`-pc_mg_levels 4`	
	`-snes_monitor -snes_view`	Describe solver

A 3D Problem

Problem has:

- 1,689,600 unknowns (on the fine level)
- 89,395,200 nonzeros

	Options	Explanation
`./ex48`	`-M 5 -N 5`	Coarse problem size
	`-da_refine 5`	5 levels of refinement
	`-ksp_rtol 1.0e-9`	Solver tolerance
	`-thi_mat_type baij`	Needs SOR
	`-pc_type mg`	4 levels of multigrid
	`-pc_mg_levels 4`	
	`-snes_monitor -snes_view`	Describe solver

Outline

Sections

Sections associate data to submeshes

- Name comes from section of a fiber bundle
  - Generalizes linear algebra paradigm
- Define `restrict()`,`update()`
- Define `complete()`
- Assembly routines take a `Sieve` and several `Section`s
  - This is called a `Bundle`

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)

AMG

Why not use AMG?

- Of course we will try AMG
  - GAMG, `-pc_type gamg`
  - ML, `-download-ml`, `-pc_type ml`
  - BoomerAMG, `-download-hypre`, `-pc_type hypre -pc_hypre_type boomeramg`
- Problems with
  - vector character
  - anisotropy
  - scalability of setup time

AMG

Why not use AMG?

- Of course we will try AMG
  - GAMG, `-pc_type gamg`
  - ML, `-download-ml`, `-pc_type ml`
  - BoomerAMG, `-download-hypre`, `-pc_type hypre -pc_hypre_type boomeramg`
- Problems with
  - vector character
  - anisotropy
  - scalability of setup time

AMG

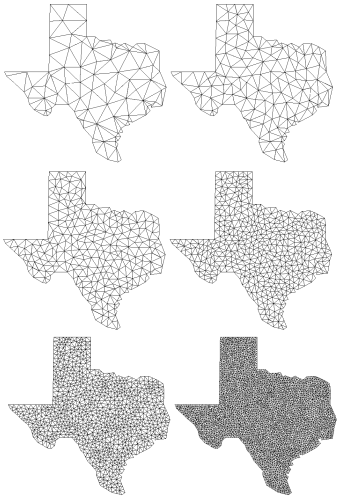
Why not use AMG?

- Of course we will try AMG
  - GAMG, `-pc_type gamg`
  - ML, `-download-ml`, `-pc_type ml`
  - BoomerAMG, `-download-hypre`, `-pc_type hypre -pc_hypre_type boomeramg`
- Problems with
  - vector character
  - anisotropy
  - scalability of setup time

Unstructured Meshes

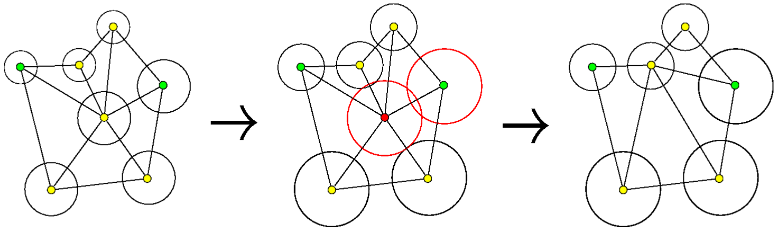
- Same `DMMG` options as the structured case
- Mesh refinement
  - Ruppert algorithm in Triangle and TetGen
- Mesh coarsening
  - Talmor-Miller algorithm in PETSc
- More advanced options
  - `-dmmg_refine`
  - `-dmmg_hierarchy`
- Current version only works for linear elements

Coarsening



- Users want to control the mesh
- Developed efficient, topological coarsening
  - Miller, Talmor, Teng algorithm
- Provably well-shaped hierarchy

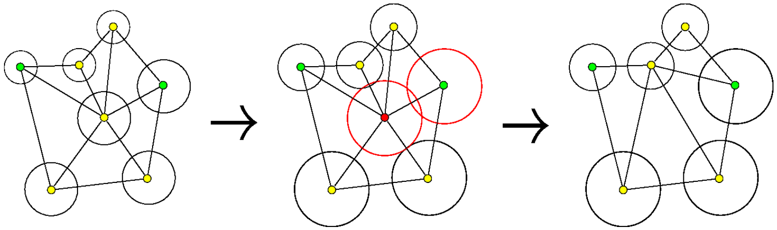
Miller-Talmor-Teng Algorithm



Simple Coarsening

1. Compute a spacing function f for the mesh (Koebe)
2. Scale f by a factor $C > 1$
3. Choose a maximal independent set of vertices for new f
4. Retriangulate

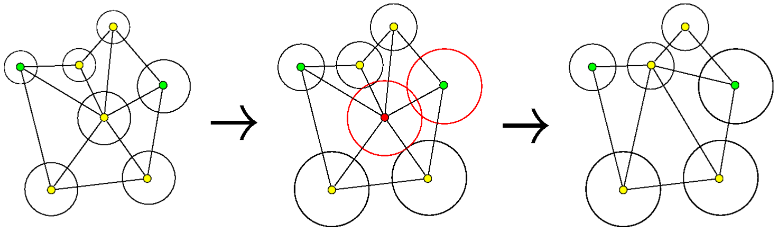
Miller-Talmor-Teng Algorithm



Simple Coarsening

1. Compute a spacing function *f* for the mesh (Koebe)
2. Scale *f* by a factor $C > 1$
3. Choose a maximal independent set of vertices for new *f*
4. Retriangulate

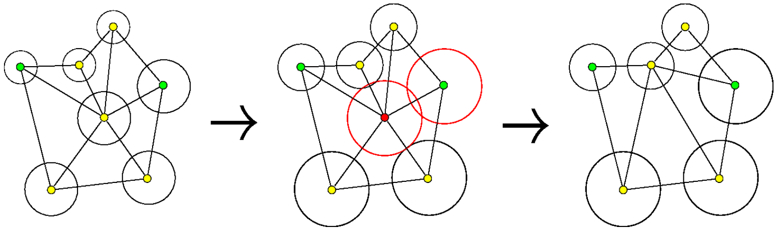
Miller-Talmor-Teng Algorithm



Simple Coarsening

1. Compute a spacing function f for the mesh (Koebe)
2. Scale f by a factor $C > 1$
3. Choose a maximal independent set of vertices for new f
4. Retriangulate

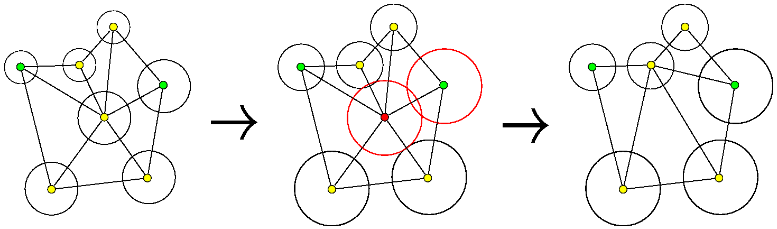
Miller-Talmor-Teng Algorithm



Simple Coarsening

1. Compute a spacing function f for the mesh (Koebe)
2. Scale f by a factor $C > 1$
3. Choose a maximal independent set of vertices for new f
4. Retriangulate

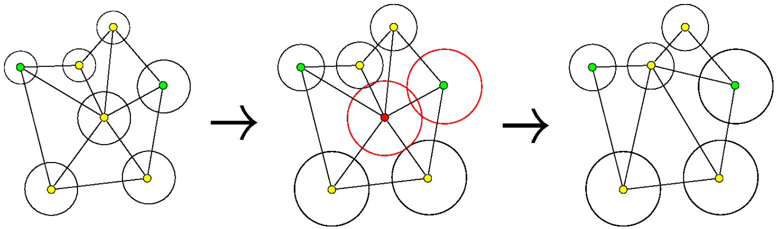
Miller-Talmor-Teng Algorithm



Caveats

1. Must generate coarsest grid in hierarchy first
2. Must choose boundary vertices first (and protect boundary)
3. Must account for boundary geometry

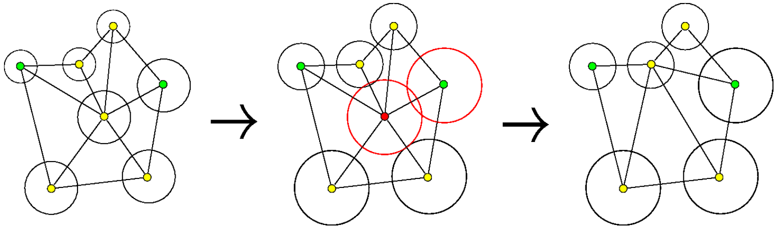
Miller-Talmor-Teng Algorithm



Caveats

1. Must generate coarsest grid in hierarchy first
2. Must choose boundary vertices first (and protect boundary)
3. Must account for boundary geometry

Miller-Talmor-Teng Algorithm

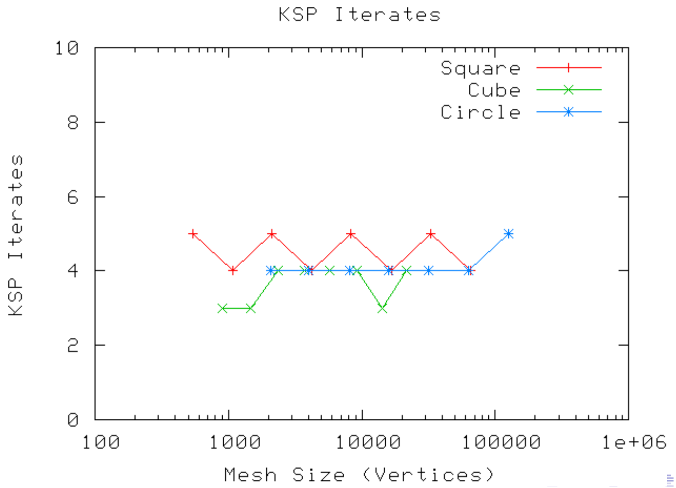


Caveats

1. Must generate coarsest grid in hierarchy first
2. Must choose boundary vertices first (and protect boundary)
3. Must account for boundary geometry

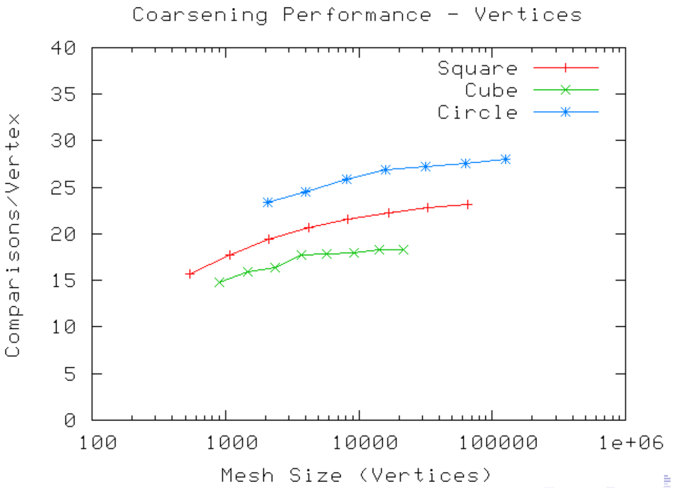
GMG Performance

For simple domains, everything works as expected:
Linear solver iterates are constant as system size increases:



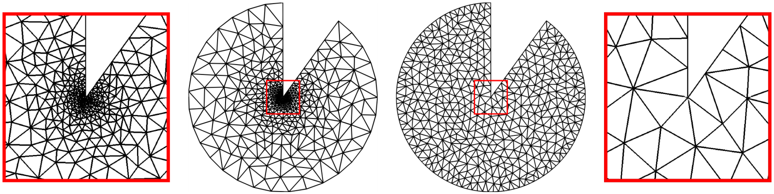
GMG Performance

For simple domains, everything works as expected:
Work to build the preconditioner is constant as system size increases:



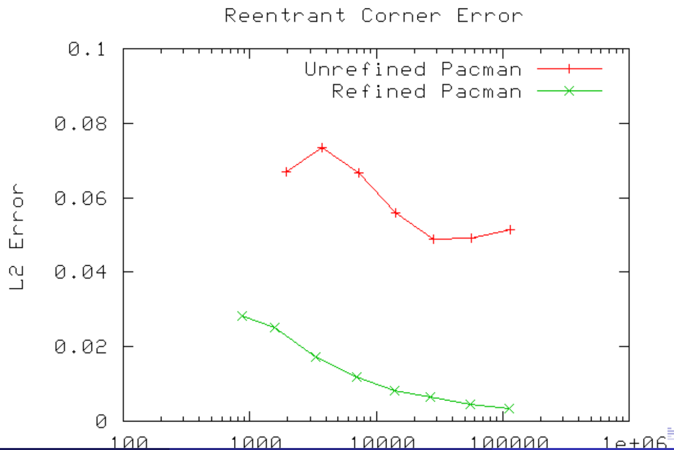
Reentrant Problems

- Reentrant corners need nonnuiform refinement to maintain accuracy
- Coarsening preserves accuracy in MG without user intervention



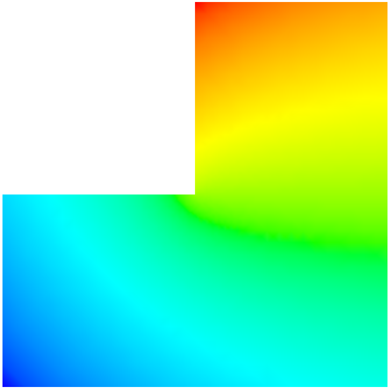
Reentrant Problems

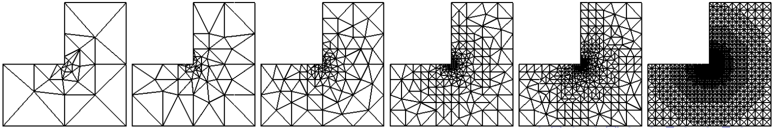
- Reentrant corners need nonnuiform refinement to maintain accuracy
- Coarsening preserves accuracy in MG without user intervention



Reentrant Problems

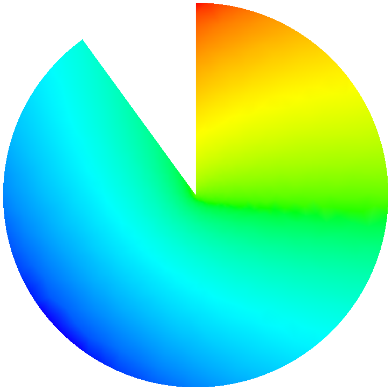
Exact Solution for reentrant problem: $u(x,y) = r^{\frac{2}{3}} sin(\frac{2}{3}\theta)$

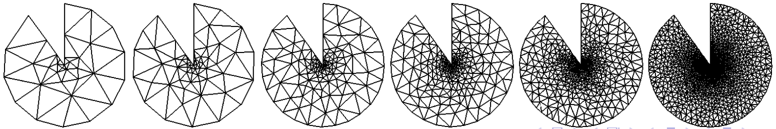




Reentrant Problems

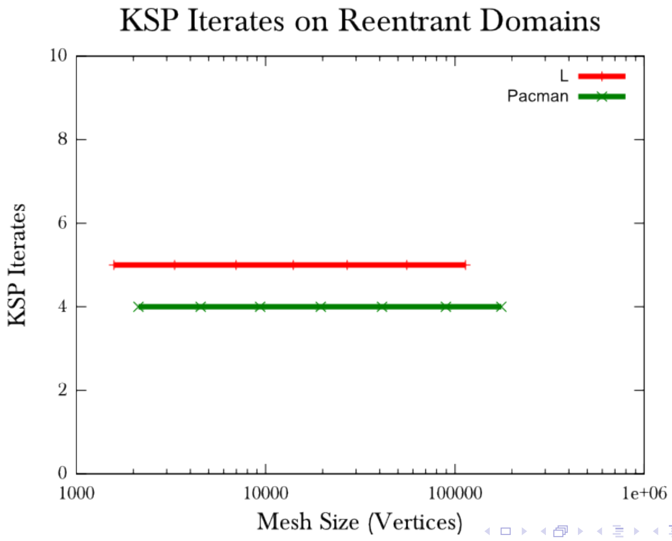
Exact Solution for reentrant problem: $u(x, y) = r^{\frac{2}{3}} sin(\frac{2}{3}\theta)$





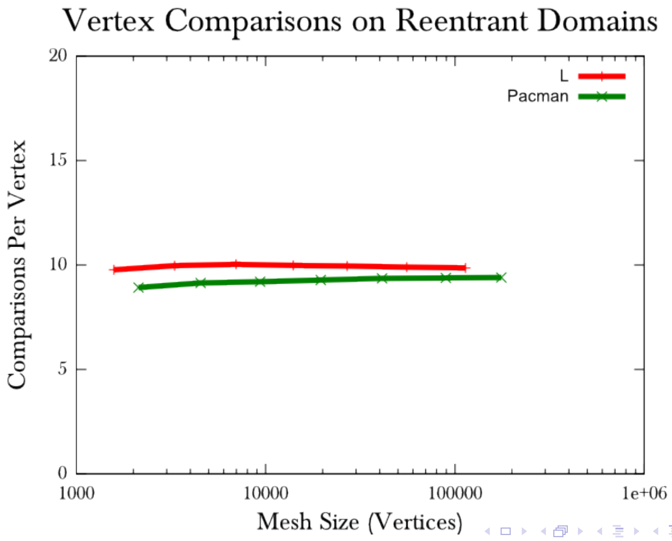
GMG Performance

Linear solver iterates are constant as system size increases:



GMG Performance

Work to build the preconditioner is constant as system size increases:



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.