The libMesh Finite Element Library

Object-Oriented High-Performance Computing

Benjamin S. Kirk[†]

benjamin.kirk@nasa.gov

John W. Peterson[‡]

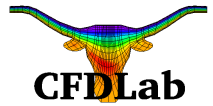
peterson@cfdlab.ae.utexas.edu

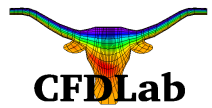
Roy H. Stogner[⋆]

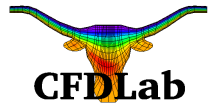
roystgnr@ices.utexas.edu

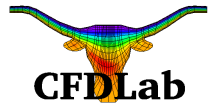
[†]NASA Lyndon B. Johnson Space Center, USA
[‡]Idaho National Labs, USA
[⋆]The University of Texas at Austin, USA

June 17, 2013









Code Contributors

Benjamin S. Kirk	benkirk
Bill Barth	bbarth
Cody Permann	permcody
Daniel Dreyer	ddreyer
David Andrs	andrsd
David Knezevic	knezed01
Derek Gaston	friedmud
Dmitry Karpeev	karpeev
Florian Prill	fprill
Jason Hales	jasondhales
John W. Peterson	jwpeterson
Paul T. Bauman	pbauman
Roy H. Stogner	roystgnr
Steffen Petersen	spetersen
Sylvain Vallaghe	svallagh
Tim Kroeger	sheep_tk
Truman Ellis	trumanellis
Wout Ruijter	woutruijter

- Thanks to Wolfgang Bangerth and the `deal.II` team for initial technical inspiration.
- Also, thanks to Jed Brown, Robert McLay, & many others for discussions over the years.

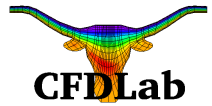
Thanks to Dr. Graham F. Carey

The original development team was heavily influenced by Professor Graham F. Carey, professor of aerospace engineering and engineering mechanics at The University of Texas at Austin, director of the ICES Computational Fluid Dynamics Laboratory, and holder of the Richard B. Curran Chair in Engineering.
Many of the technologies employed in libMesh were implemented because Dr. Carey taught them to us, we went back to the lab, and immediately began coding. In a very real way, he was ultimately responsible for this library that we hope you may find useful, despite his continued insistence that "no one ever got a PhD from here for writing a code."

CFDLab

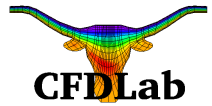
Background

- Modern simulation software is complex:
  - Implicit numerical methods
  - Massively parallel computers
  - Adaptive methods
  - Multiple, coupled physical processes
- There are a host of existing software libraries that excel at treating various aspects of this complexity.
- Leveraging existing software whenever possible is the most efficient way to manage this complexity.



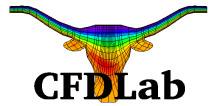
Background

- Modern simulation software is multidisciplinary:
  - Physical Sciences
  - Engineering
  - Computer Science
  - Applied Mathematics
  - ...
- It is not reasonable to expect a single person to have all the necessary skills for developing & implementing high-performance numerical algorithms on modern computing architectures.
- Teaming is a prerequisite for success.



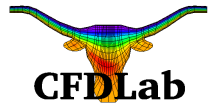
Background

- A large class of problems are amenable to mesh based simulation techniques.
- Consider some of the major components such a simulation:



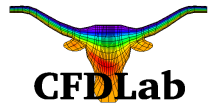
Background

- A large class of problems are amenable to mesh based simulation techniques.
- Consider some of the major components such a simulation:
  1. Read the mesh from file
  2. Initialize data structures
  3. Construct a discrete representation of the governing equations
  4. Solve the discrete system
  5. Write out results
  6. Optionally estimate error, refine the mesh, and repeat



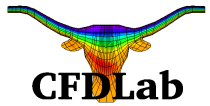
Background

- A large class of problems are amenable to mesh based simulation techniques.
- Consider some of the major components such a simulation:
  1. Read the mesh from file
  2. Initialize data structures
  3. Construct a discrete representation of the governing equations
  4. Solve the discrete system
  5. Write out results
  6. Optionally estimate error, refine the mesh, and repeat
- With the exception of step 3, the rest is *independent* of the class of problems being solved.



Background

- A large class of problems are amenable to mesh based simulation techniques.
- Consider some of the major components such a simulation:
  1. Read the mesh from file
  2. Initialize data structures
  3. Construct a discrete representation of the governing equations
  4. Solve the discrete system
  5. Write out results
  6. Optionally estimate error, refine the mesh, and repeat
- With the exception of step 3, the rest is *independent* of the class of problems being solved.
- This allows the major components of such a simulation to be abstracted & implemented in a reusable software library.



The libMesh Software Library

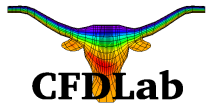
- In 2002, the libMesh library began with these ideas in mind.
- Primary goal is to provide data structures and algorithms that can be shared by disparate physical applications, that may need some combination of
  - Implicit numerical methods
  - Adaptive mesh refinement techniques
  - Parallel computing
- Unifying theme: mesh-based simulation of partial differential equations (PDEs).

CFDLab

The libMesh Software Library

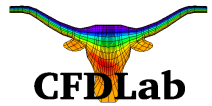
Key Point

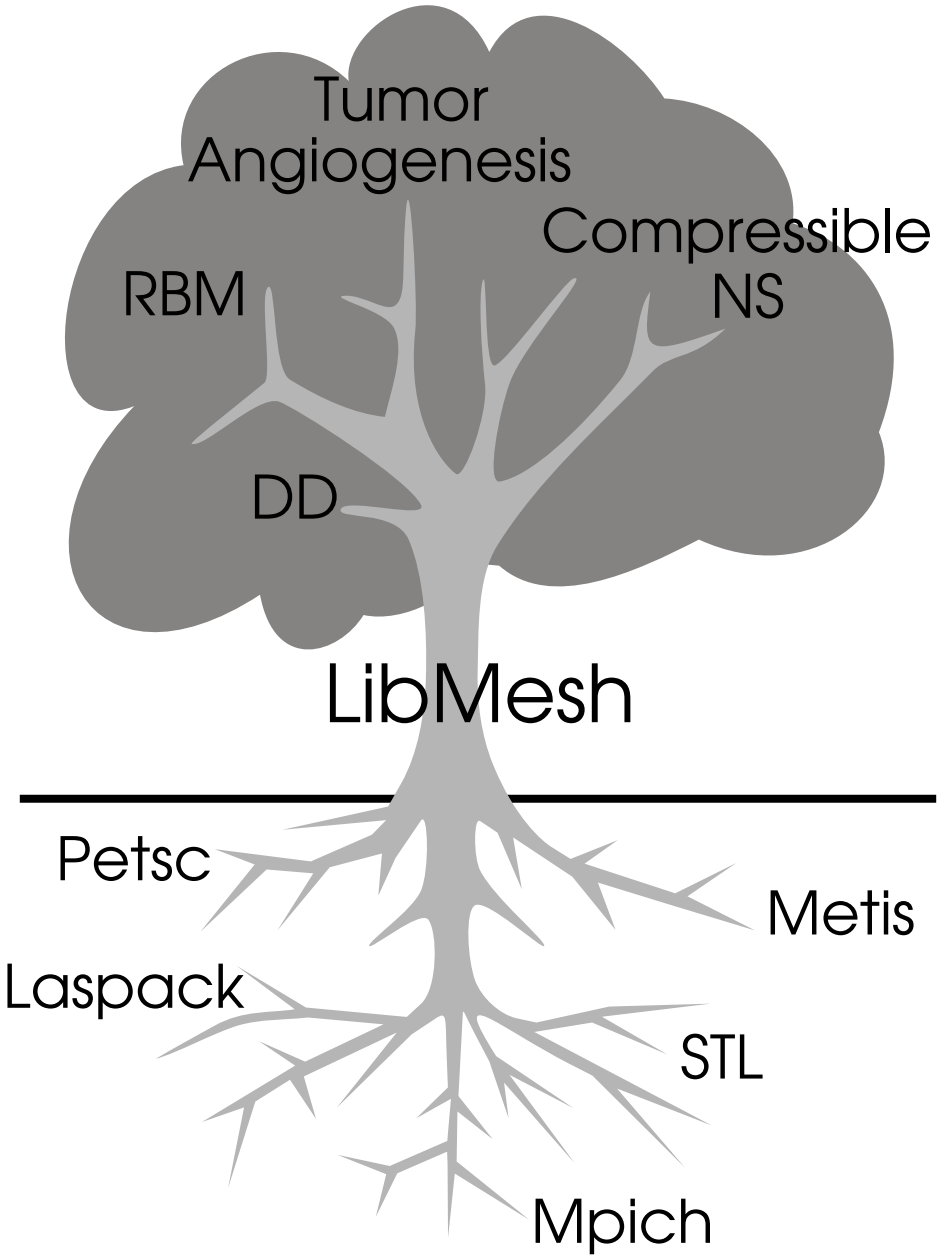
- The libMesh library is designed to be used by students, researchers, scientists, and engineers as a tool for developing simulation codes or as a tool for rapidly implementing a numerical method.
- libMesh is not an application code.
- It does not "solve problem XYZ."
  - It can be used to help you develop an application to solve problem XYZ, and to do so quickly with advanced numerical algorithms on high-performance computing platforms.



Software Reusability

- At the inception of `libMesh` in 2002, there were many high-quality software libraries that implemented some aspect of the end-to-end PDE simulation process:
  - Parallel linear algebra
  - Partitioning algorithms for domain decomposition
  - Visualization formats
  - ...
- A design goal of `libMesh` has always been to provide flexible & extensible interfaces to existing software whenever possible.
- We implement the "glue" to these pieces, as well as what we viewed as the missing infrastructure:
  - Flexible data structures for the discretization of spatial domains and systems of PDEs posed on these domains.



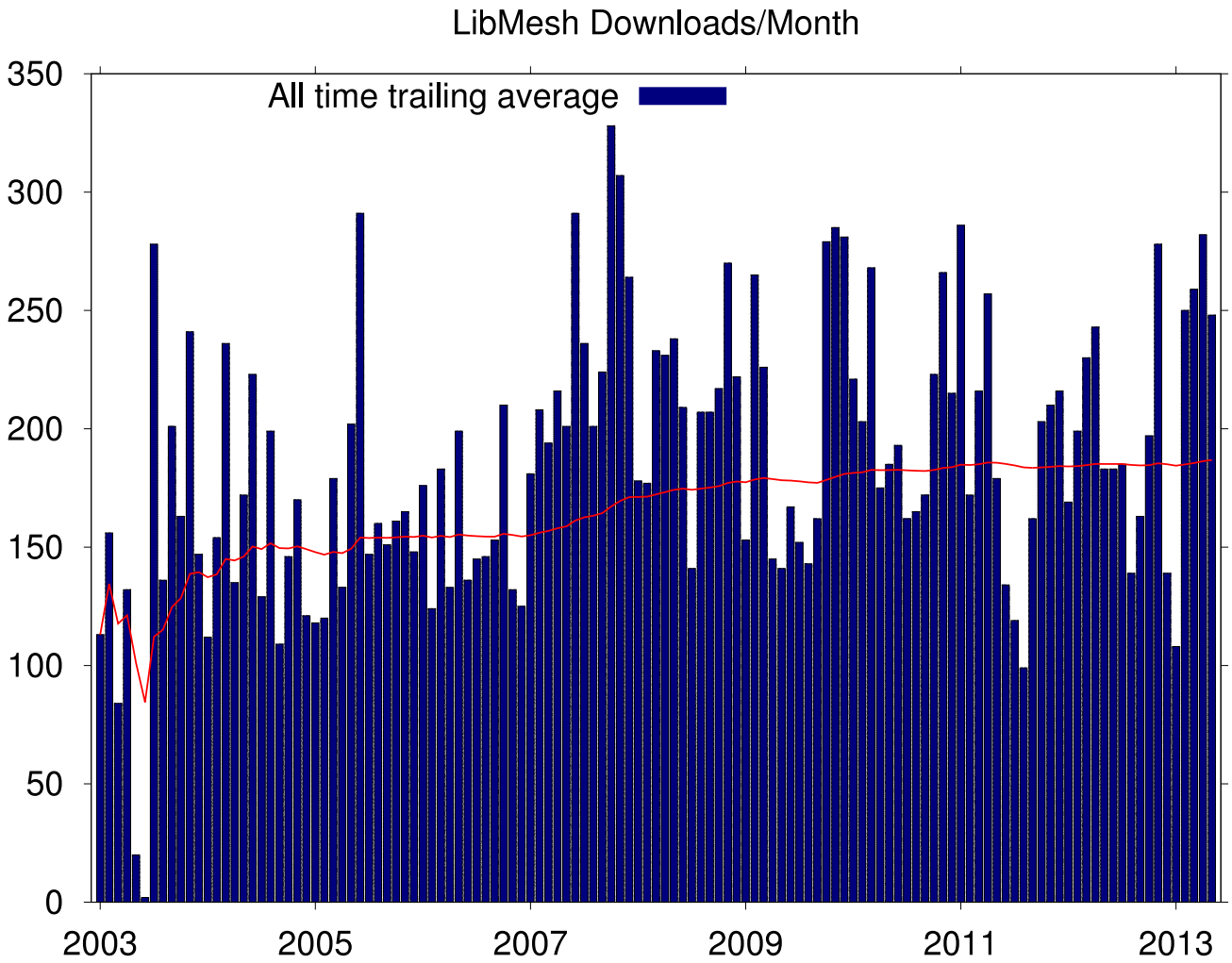


Library Structure

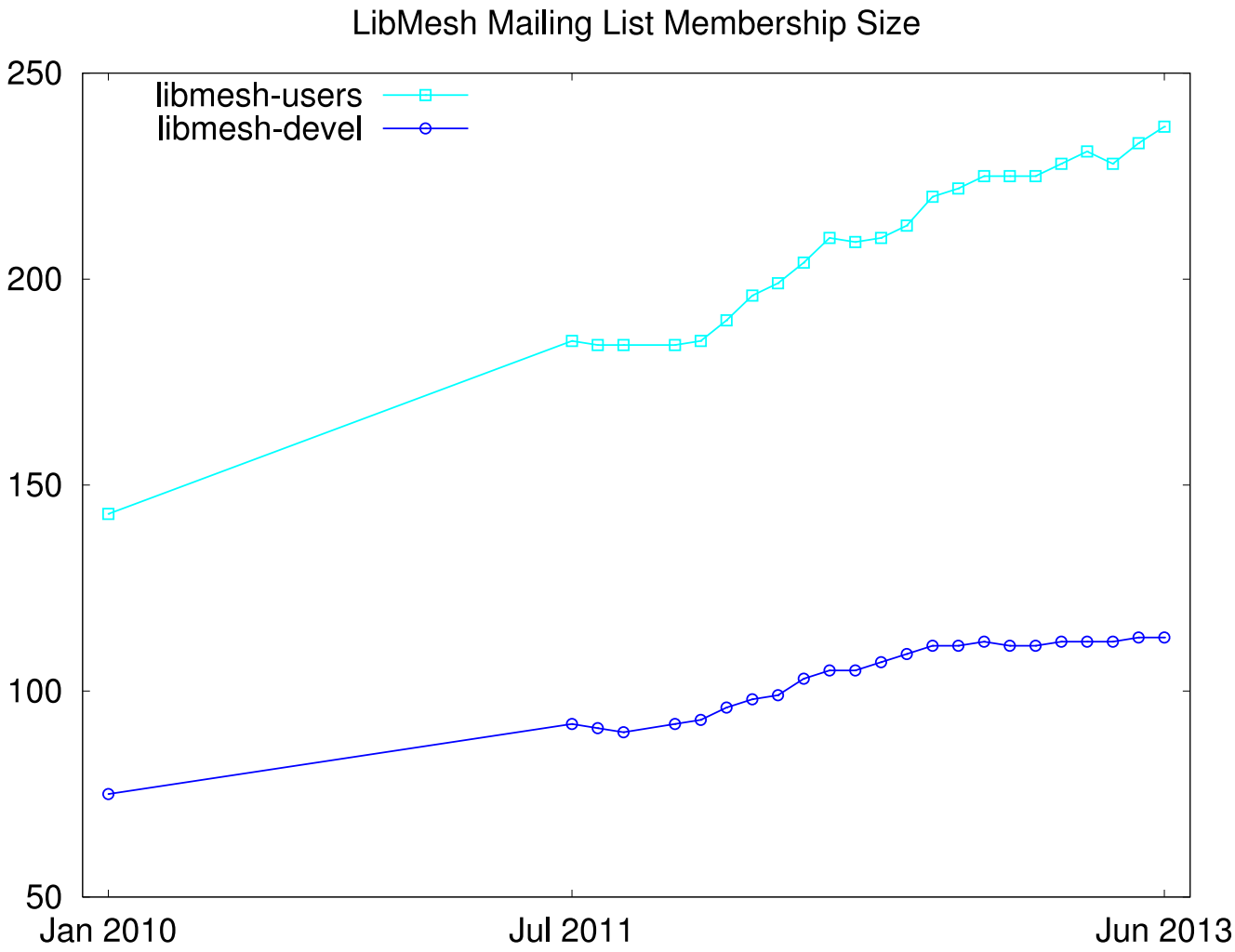
- Basic libraries are `libMesh`'s "roots"
- Application "branches" built off the library "trunk"

CFDLab

Trivia – Downloads

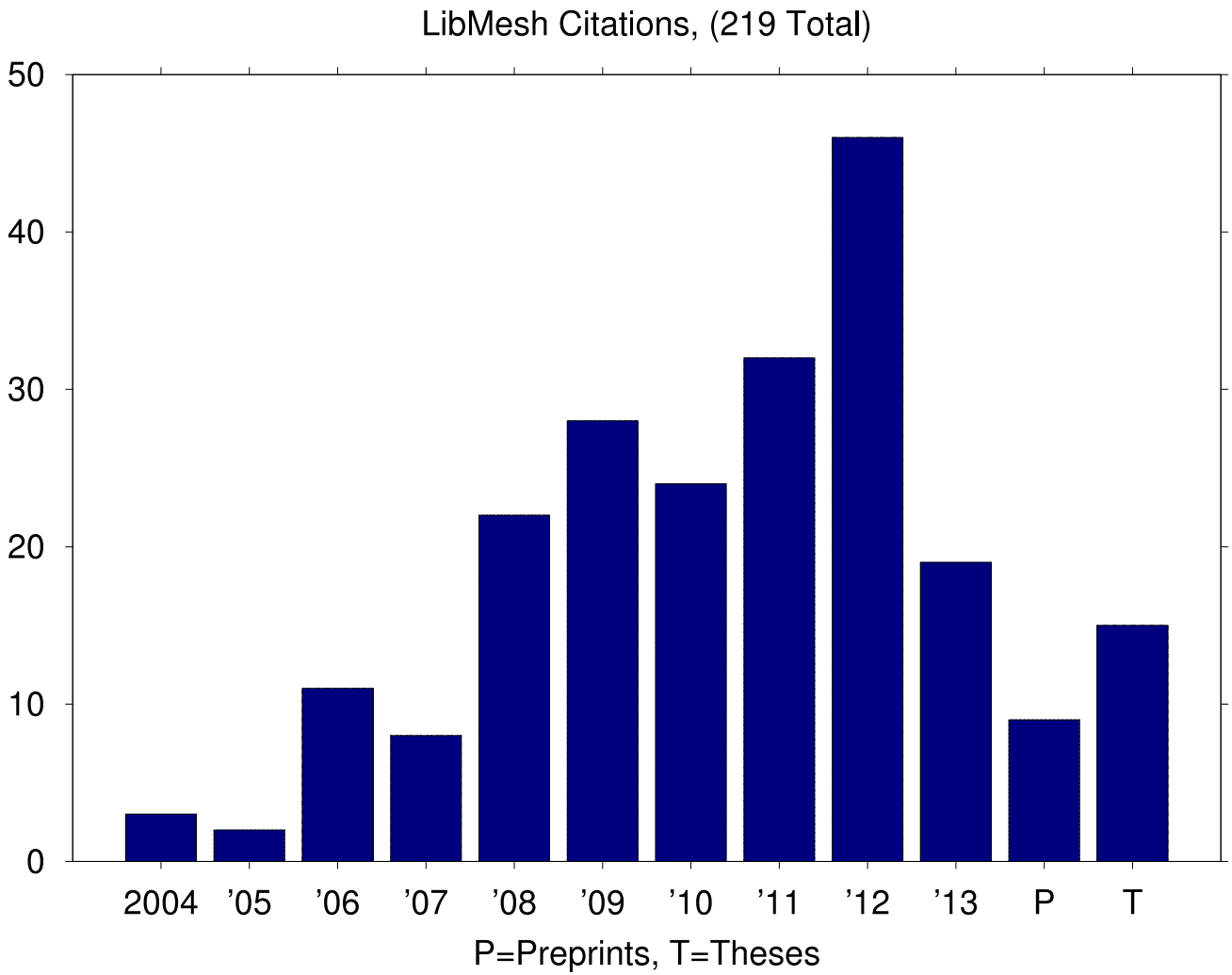


Trivia – Mailing List Membership



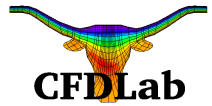
libmesh-users@lists.sourceforge.net
libmesh-devel@lists.sourceforge.net

Trivia – Citations



The “Glue”

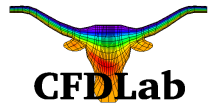
- The C++ programming language provides a powerful abstraction mechanism for separating a software interface from its implementation.
- The notion of Base Classes defining an abstract interface and Derived Classes implementing the interface is key to this programming model.



The "Glue"

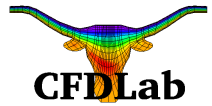
- The C++ programming language provides a powerful abstraction mechanism for separating a software interface from its implementation.
- The notion of Base Classes defining an abstract interface and Derived Classes implementing the interface is key to this programming model.
- The classic C++ example: Shapes.

```
int main ()
{
  /* using shapes polymorphically */
  Shape * shapes[2];
  shapes[0] = new Rectangle (10, 20, 5, 6);
  shapes[1] = new Circle (15, 25, 8);

  for (int i=0; i<2; ++i)
    shapes[i]->Draw();
  ...
}
```



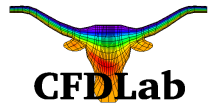
Abstract Shape

```
/* abstract interface declaration */
class Shape
{
public:
  virtual void Draw () = 0;
  virtual void MoveTo (int newx, int newy) = 0;
  virtual void RMoveTo (int dx, int dy) = 0;
};
```



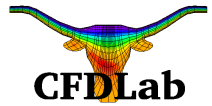
Specific Shape: Rectangle

```
/* Class Rectangle */
class Rectangle : public Shape
{
public:
  Rectangle (int x, int y, int w, int h);
  virtual void Draw ();
  virtual void MoveTo (int newx, int newy);
  virtual void RMoveTo (int dx, int dy);
  virtual void SetWidth (int newWidth);
  virtual void SetHeight (int newHeight);

private:
  int x, y;
  int width;
  int height;
};
```



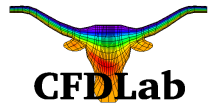
Specific Shape: Circle

```
/* Class Circle */
class Circle : public Shape
{
public:
  Circle (int initx, int inity, int initr);
  virtual void Draw ();
  virtual void MoveTo (int newx, int newy);
  virtual void RMoveTo (int dx, int dy);
  virtual void SetRadius (int newRadius);

private:
  int x, y;
  int radius;
};
```

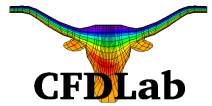


Object Polymorphism

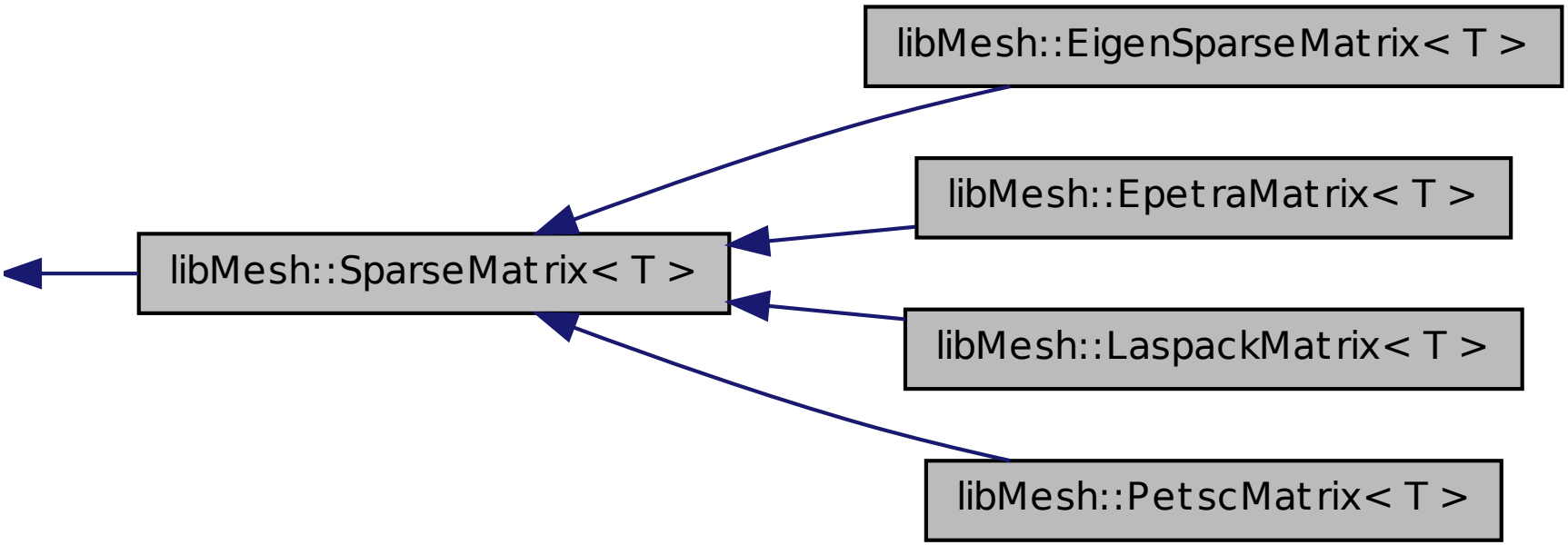
```
int main ()
{
  /* using shapes polymorphically */
  Shape * shapes[2];
  shapes[0] = new Rectangle (10, 20, 5, 6);
  shapes[1] = new Circle (15, 25, 8);

  for (int i=0; i<2; ++i)
    {
      shapes[i]->Draw();
      DoSomethingWithShape (shapes[i]);
    }

  /* access a rectangle specific function */
  Rectangle rect(0, 0, 15, 15);
  rect.SetWidth (30);
  rect.Draw ();
  ...
}
```



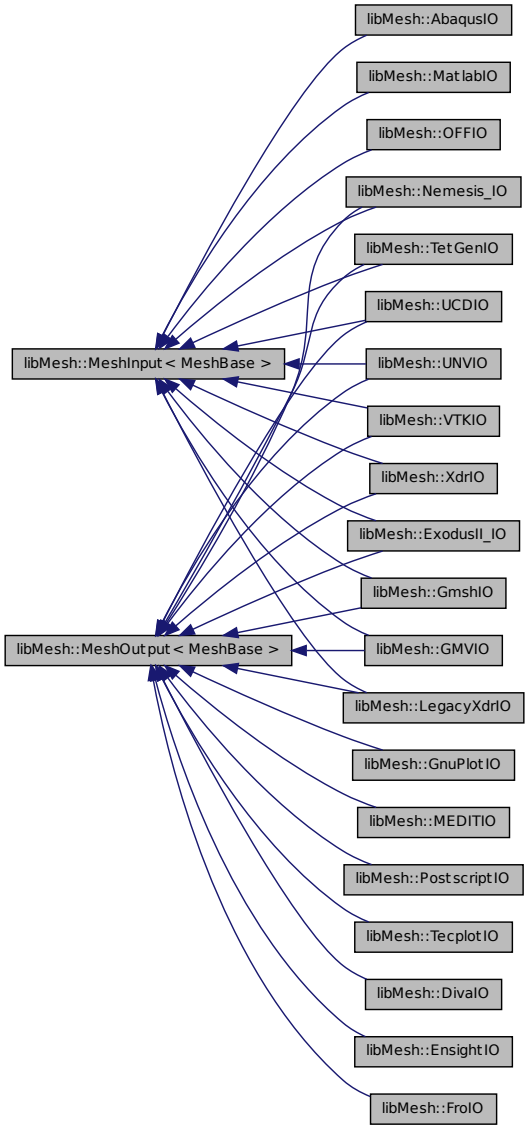
Examples of Polymorphism in `libMesh`



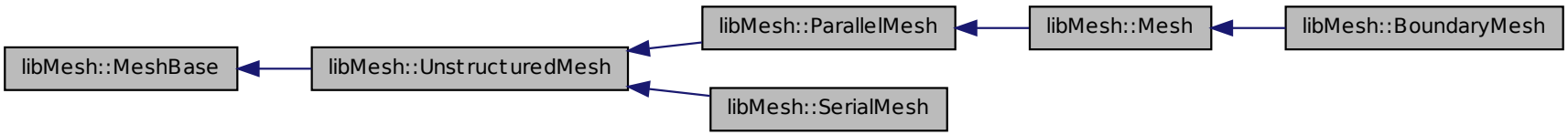
The “Glue:” Linear Algebra



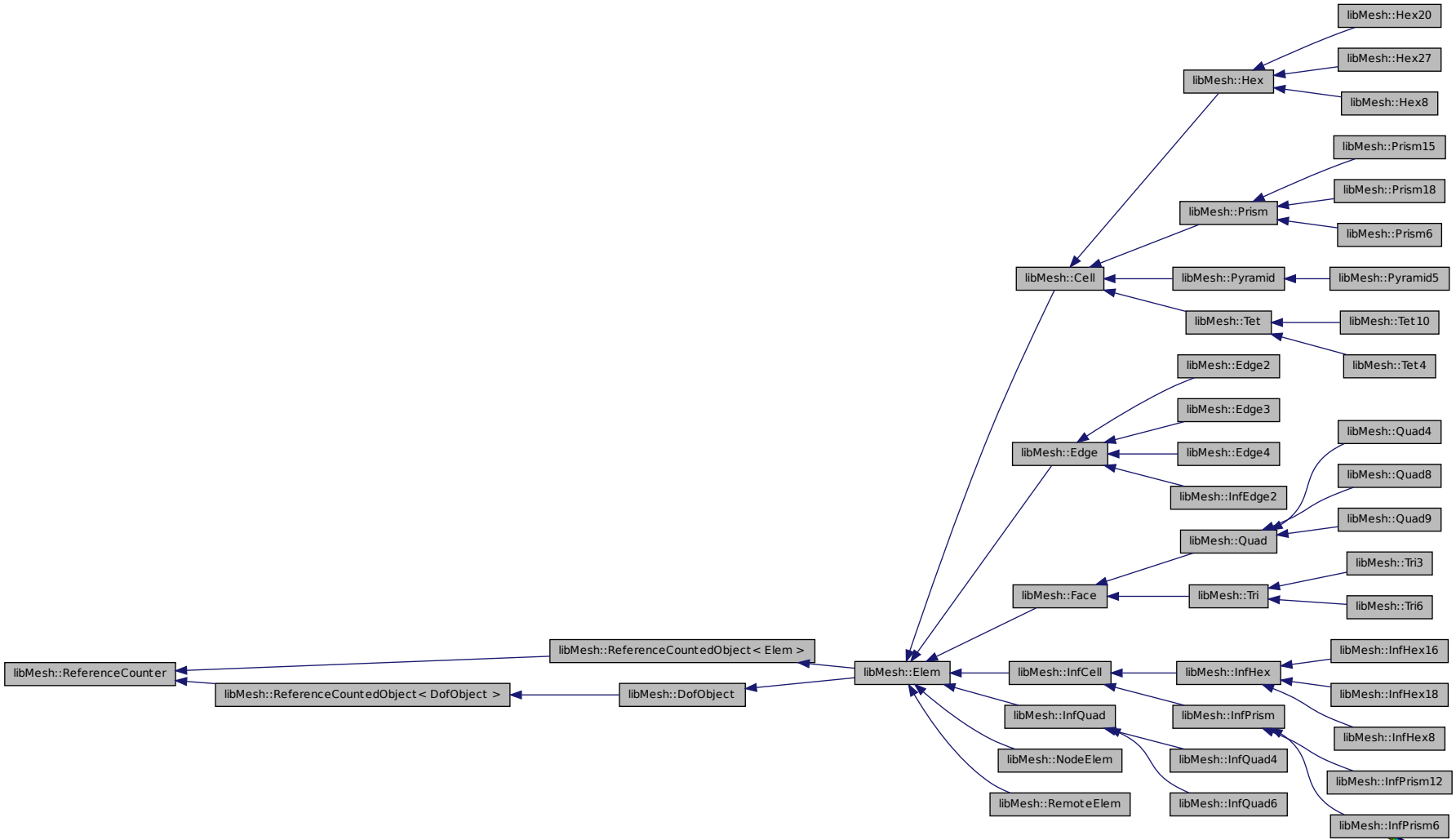
The “Glue:” I/O formats



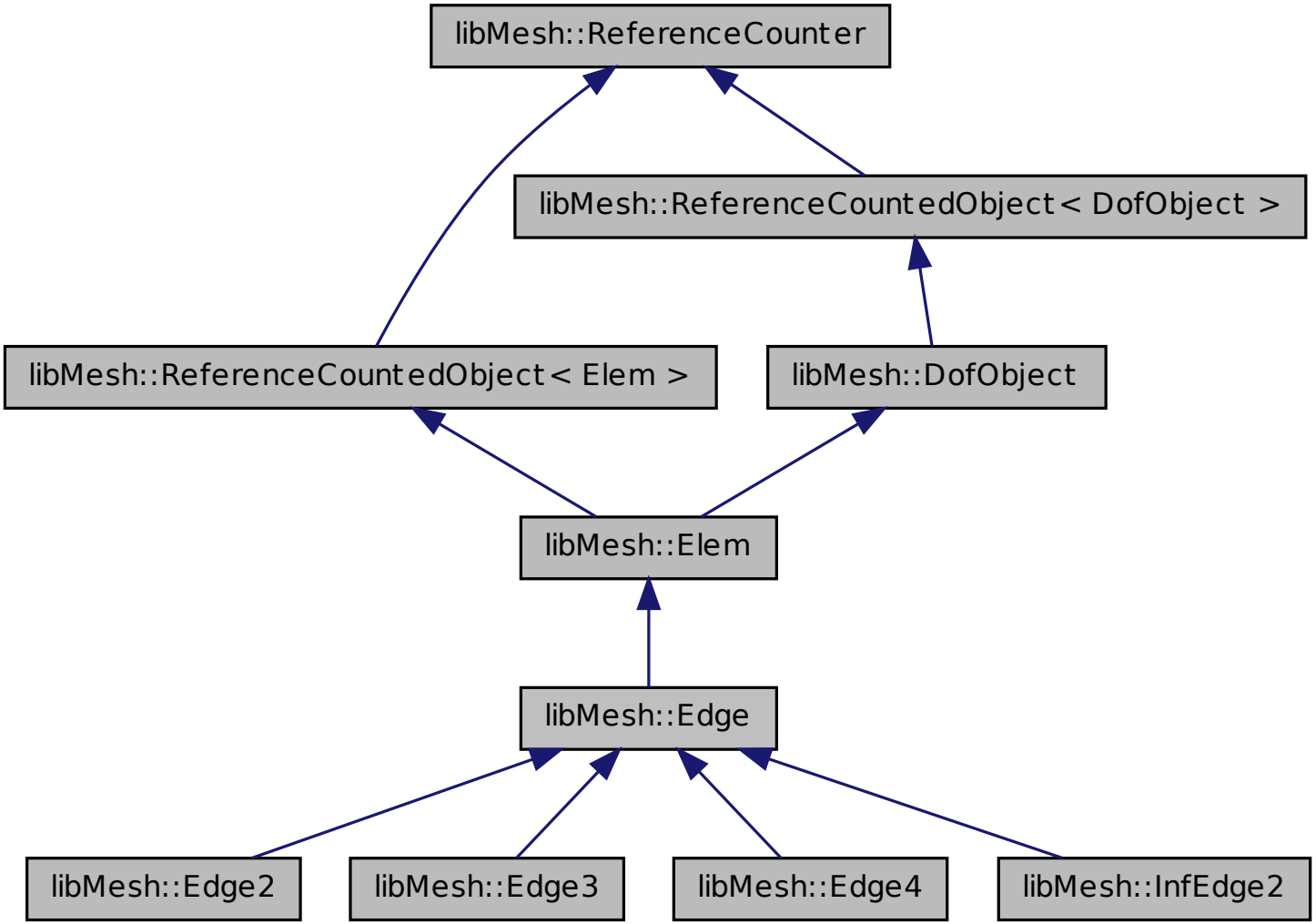
Disretization: The Mesh



Disretization: Geometric Elements

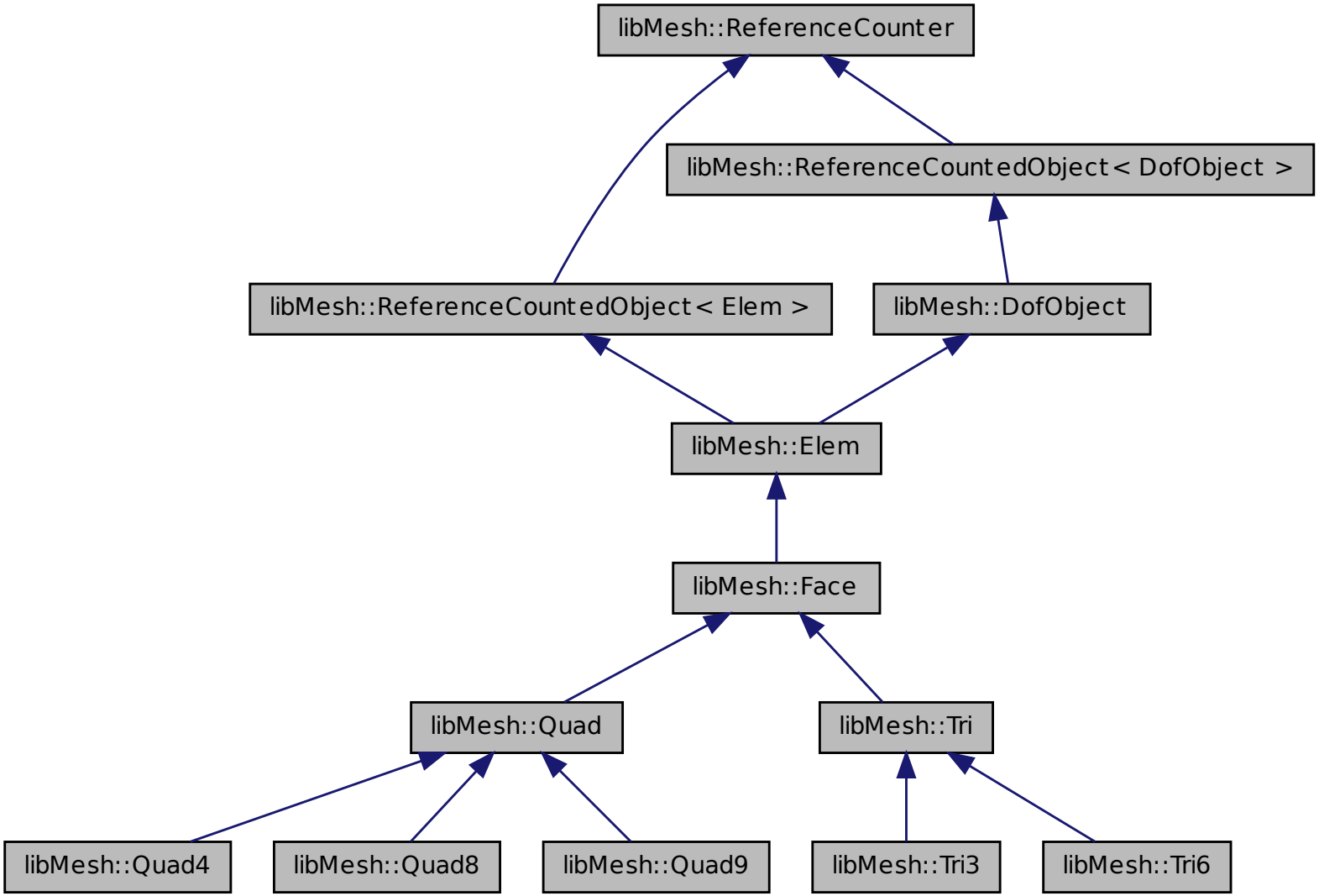


Disretization: Geometric Elements



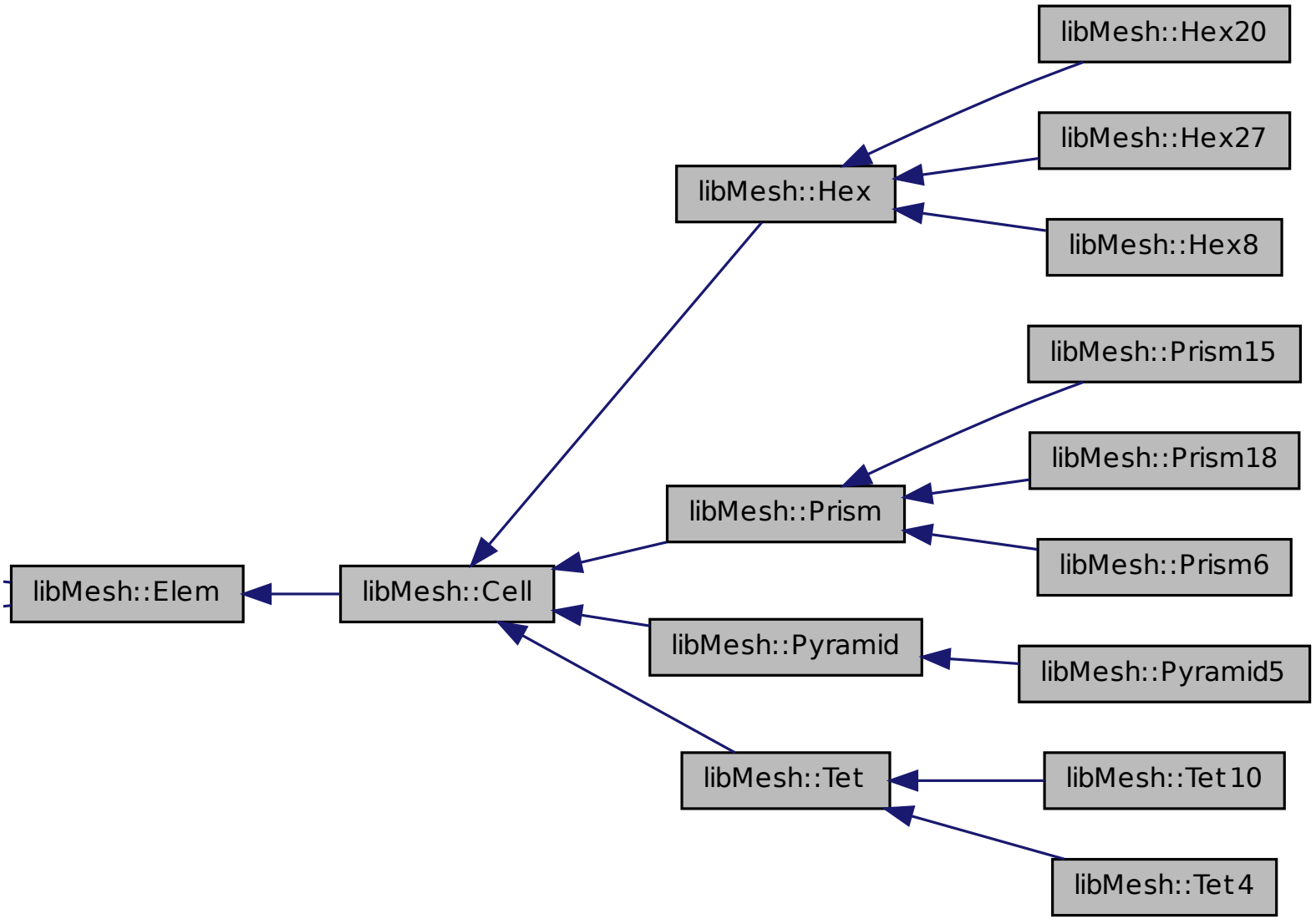
CFDLab

Disretization: Geometric Elements



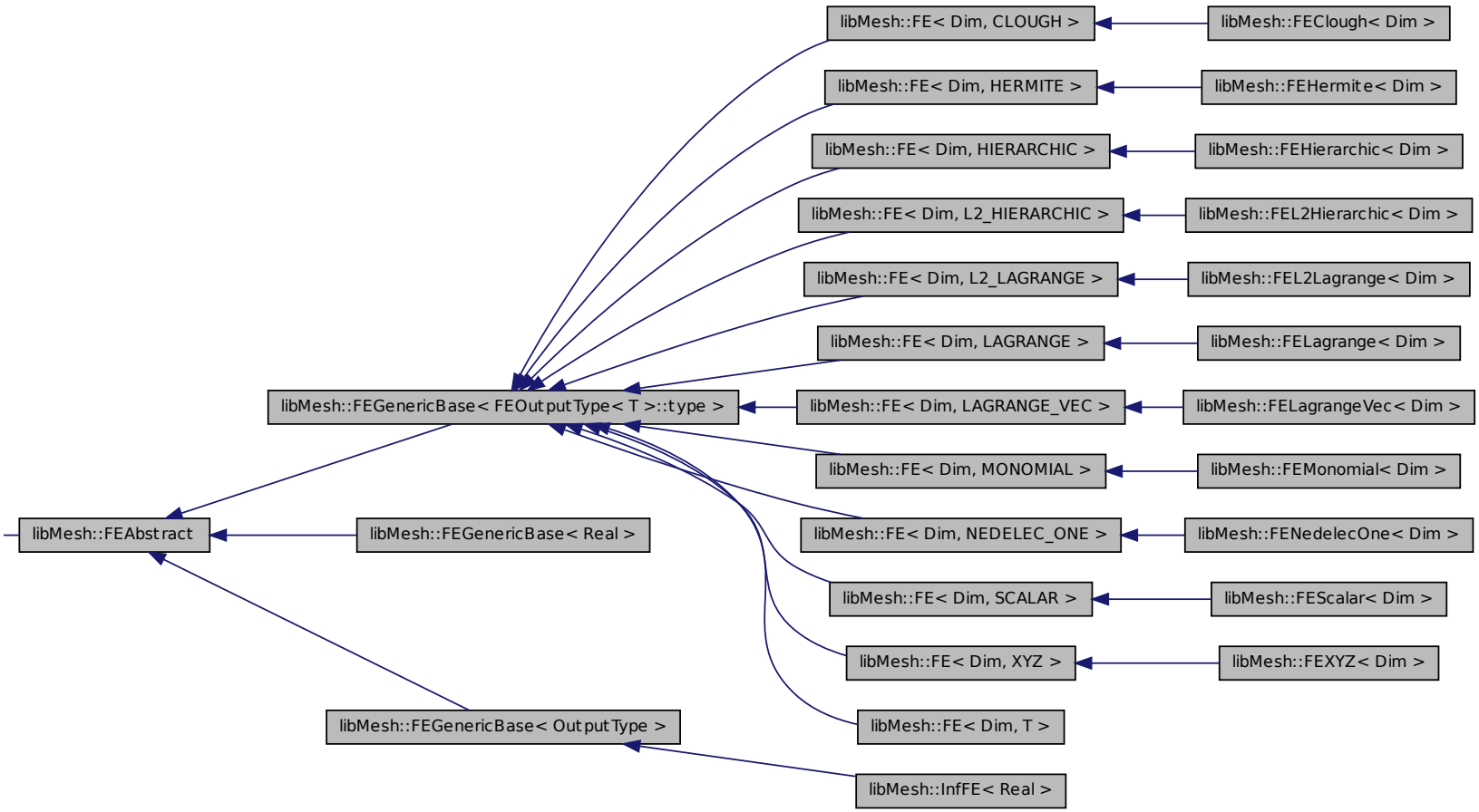
CFDLab

Disretization: Geometric Elements

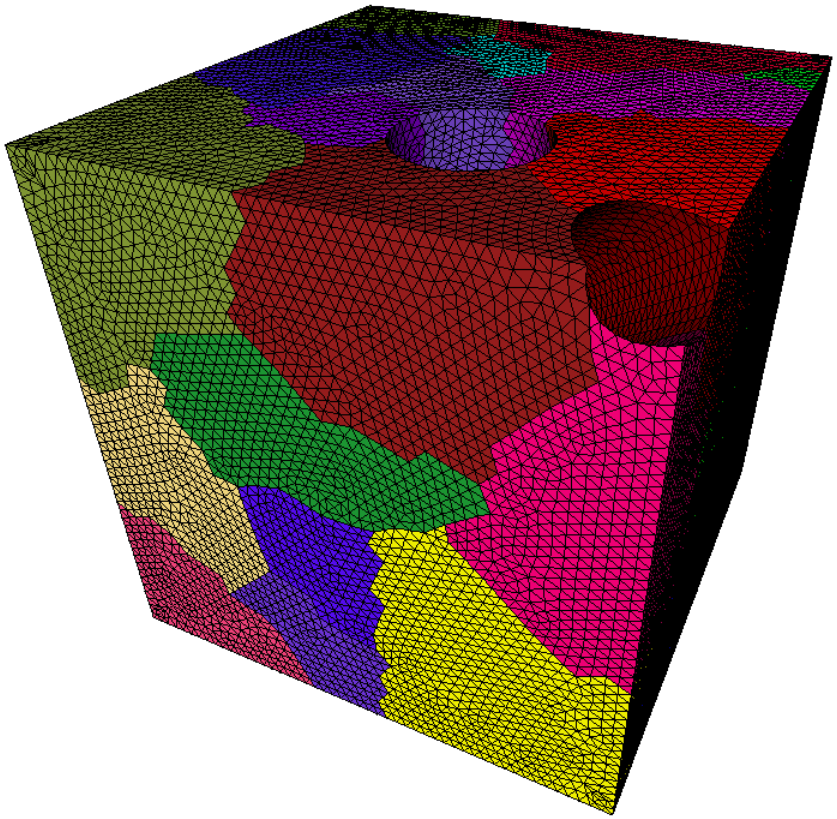


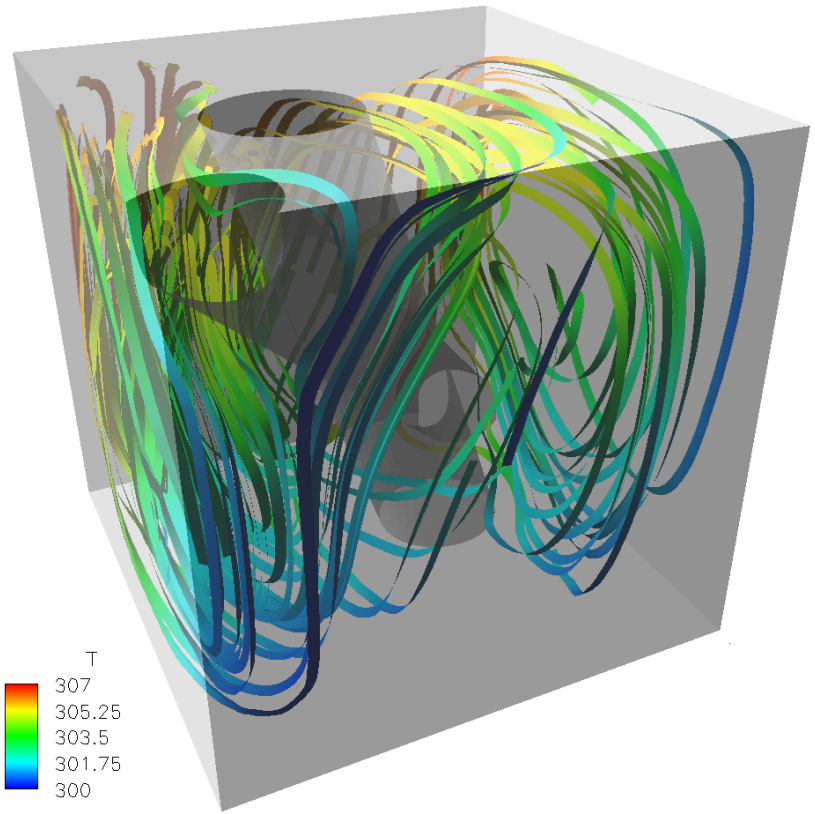
CFDLab

Disretization: Finite Elements



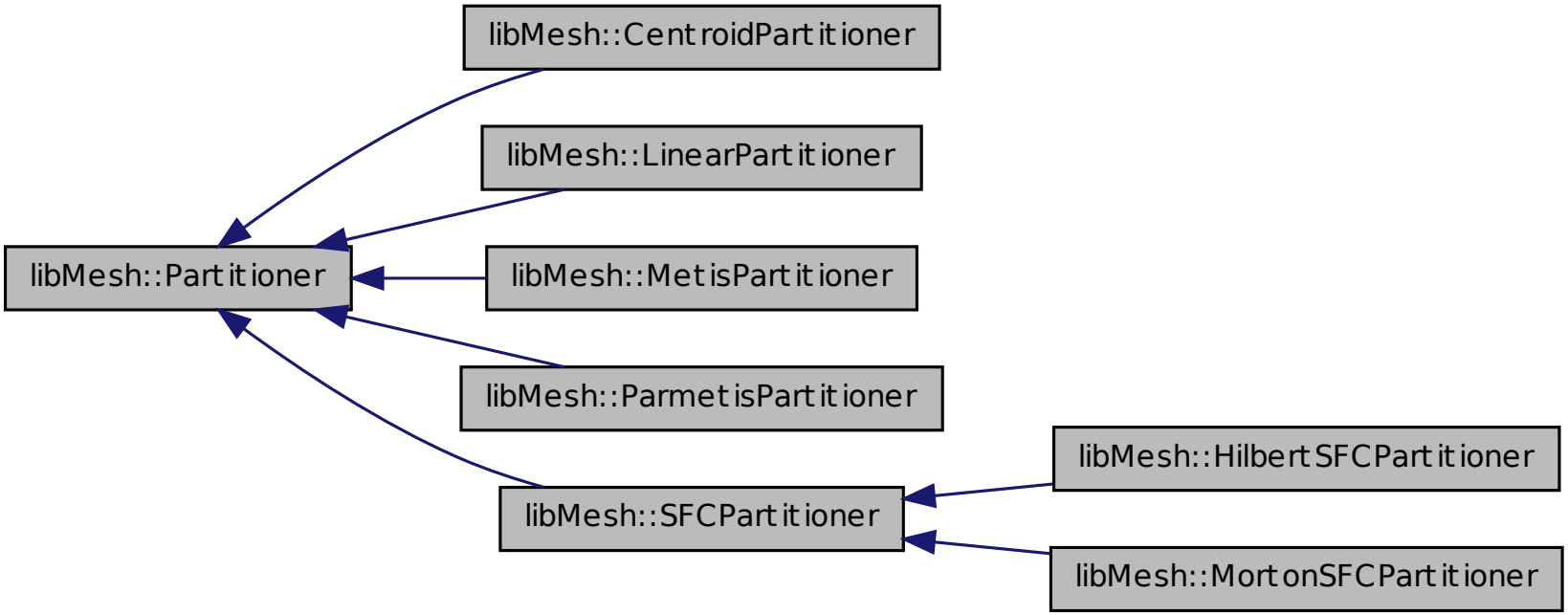
Algorithms: Domain Partitioning





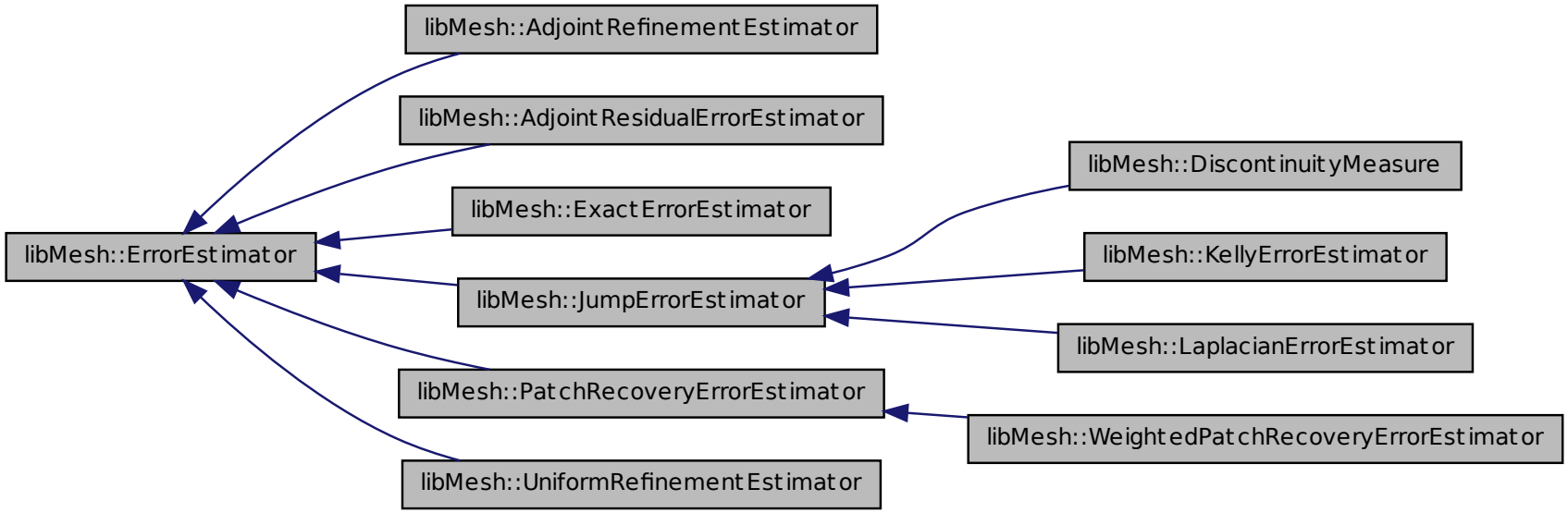
CFDLab

Algorithms: Domain Partitioning

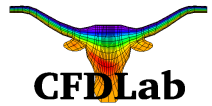


CFDLab

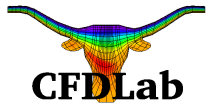
Algorithms: Error Estimation



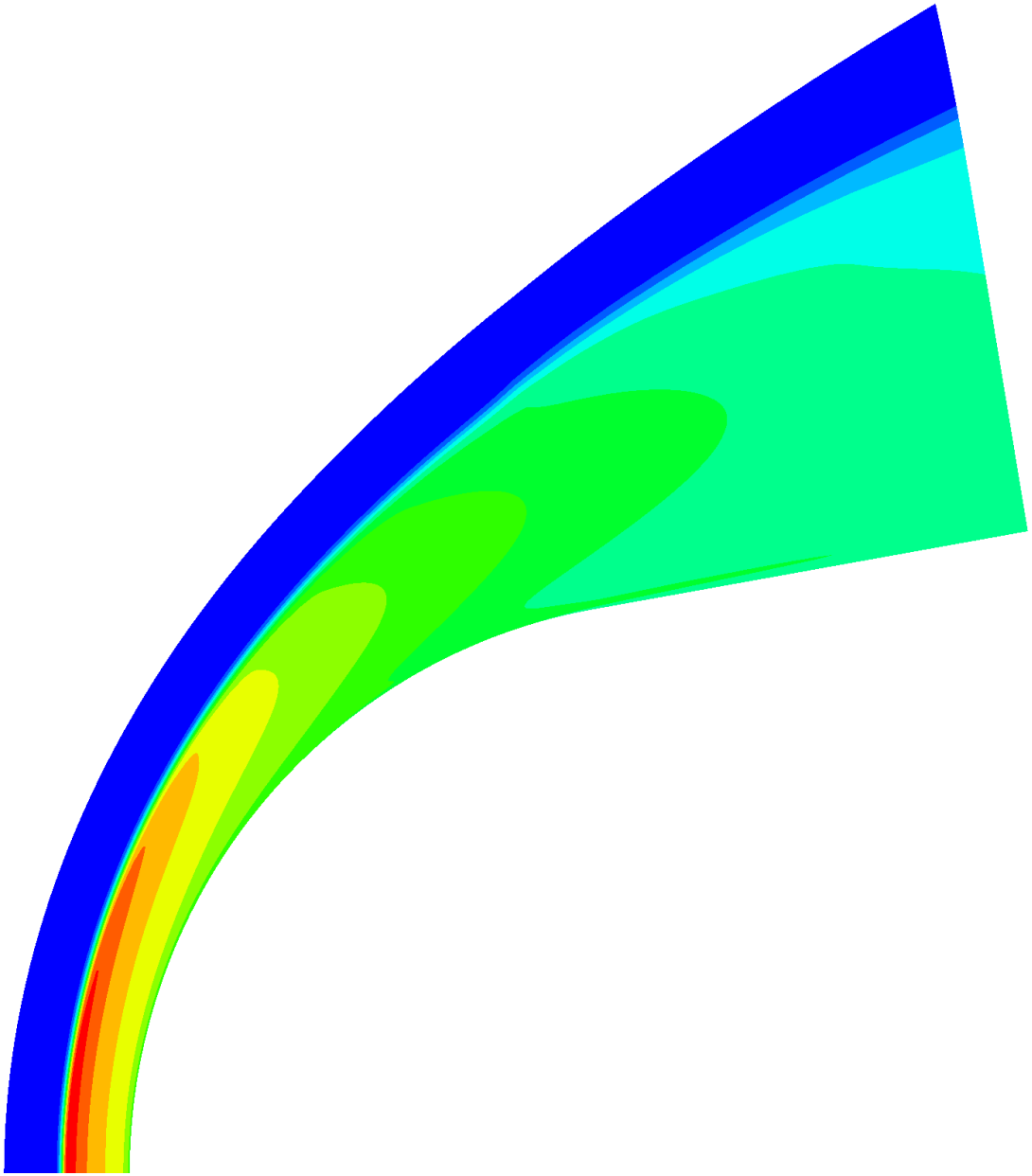
CFDLab



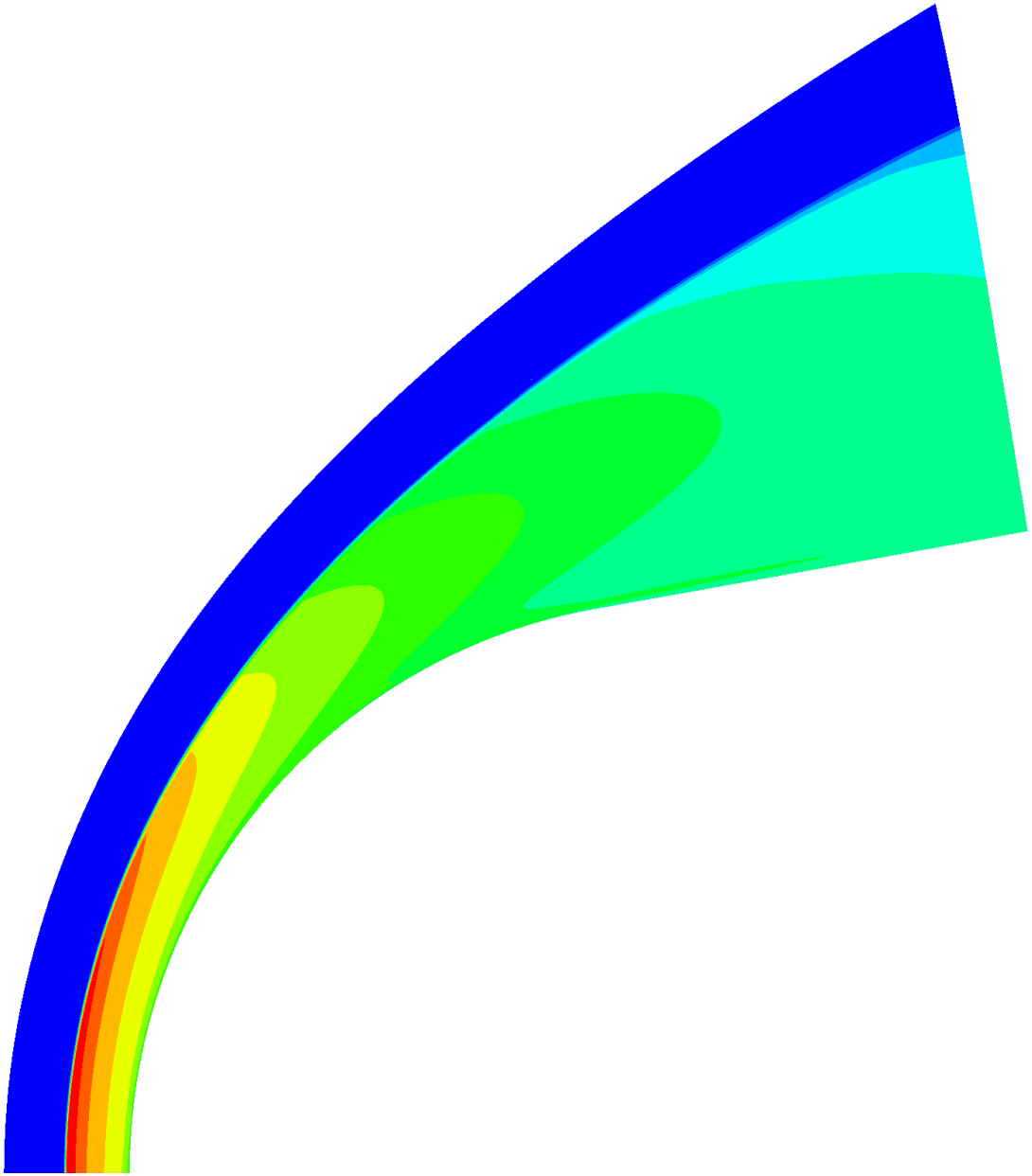
Results from Physics Applications built on top of **libMesh**



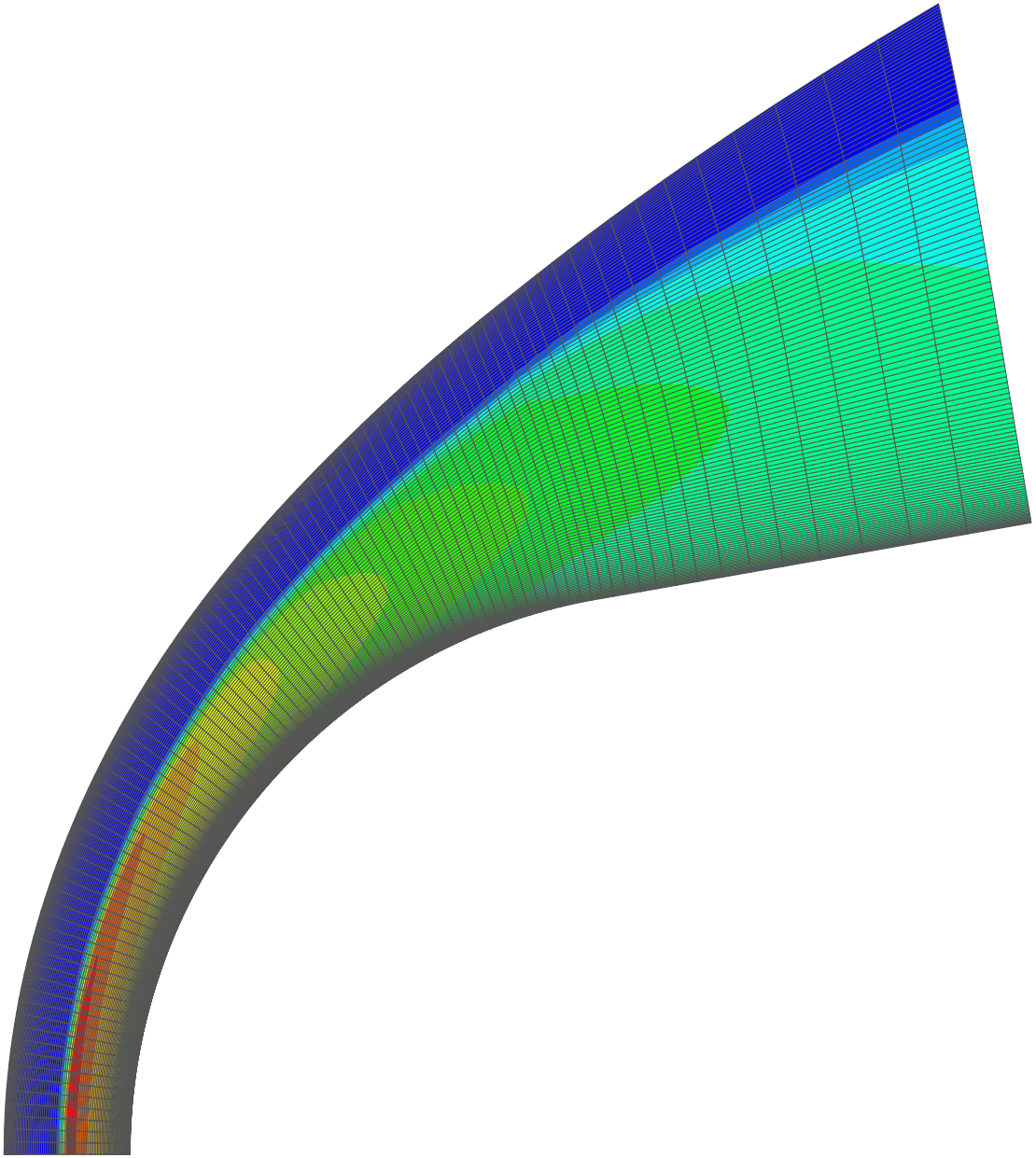
Compressible Navier-Stokes



Compressible Navier-Stokes

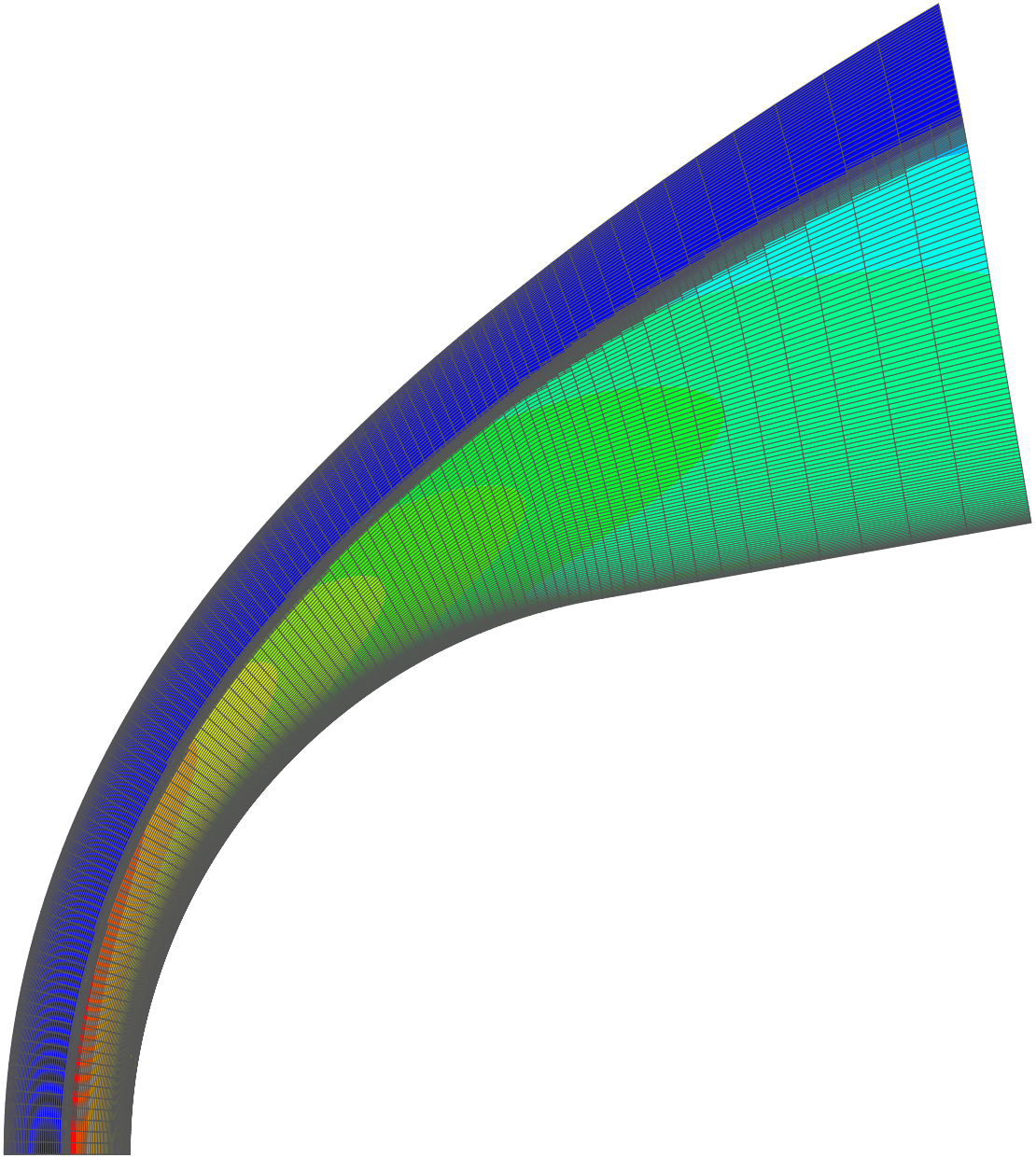


Compressible Navier-Stokes



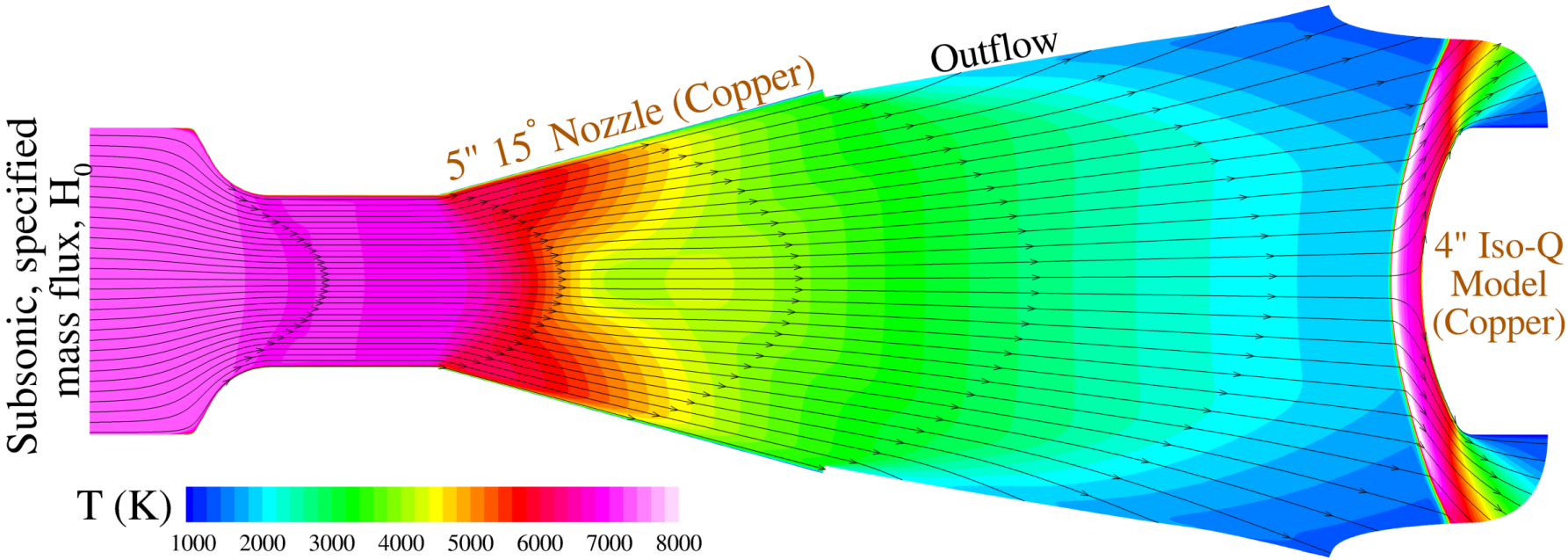
CFDLab

Compressible Navier-Stokes



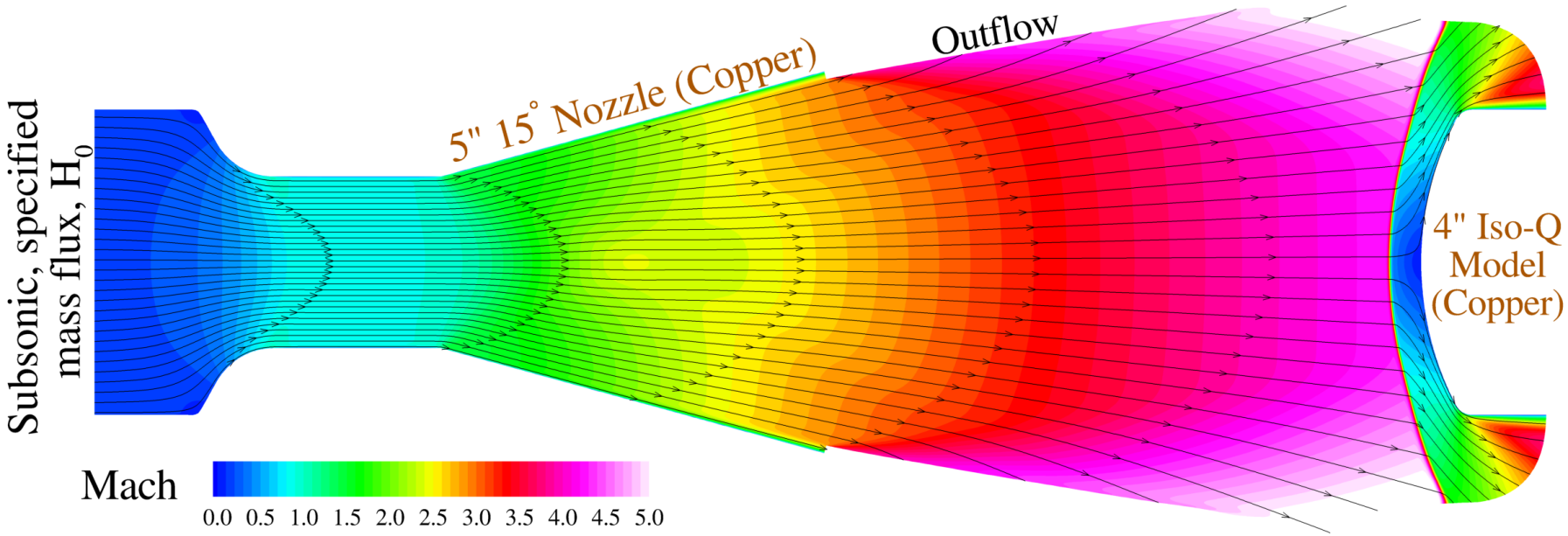
CFDLab

Arcjet Nozzle Calculation



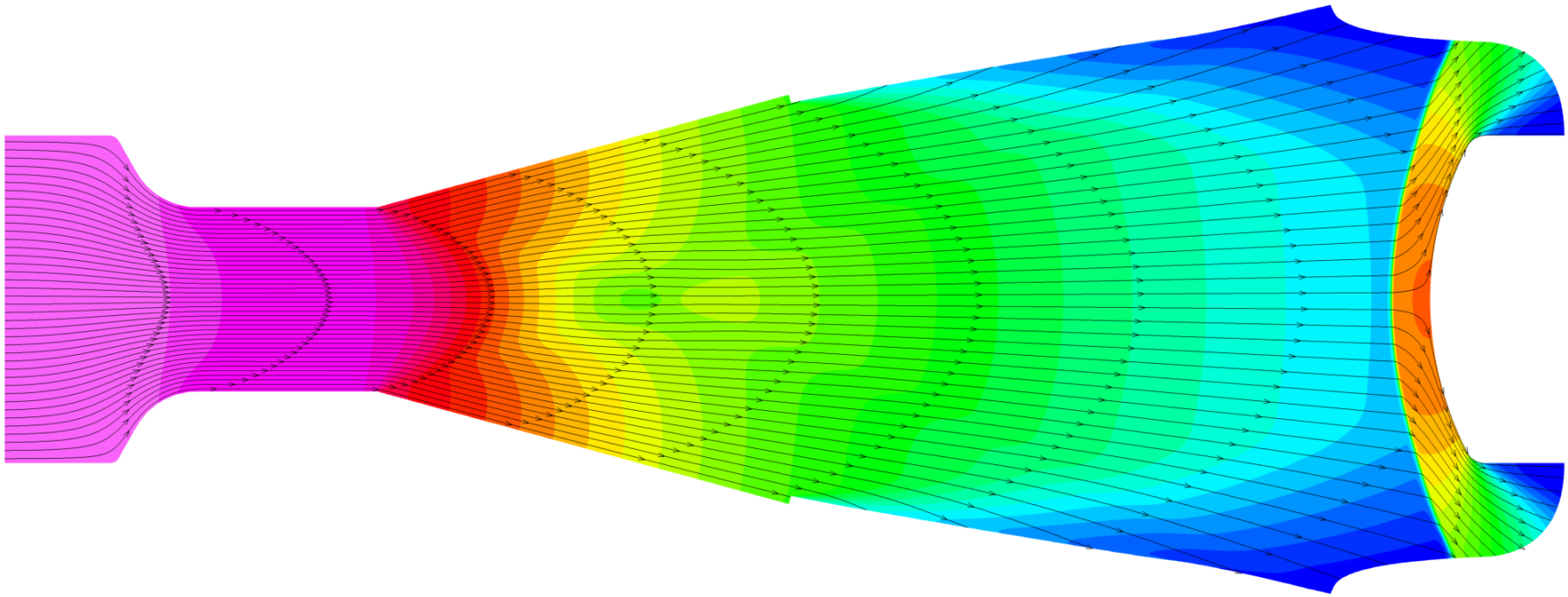
CFDLab

Arcjet Nozzle Calculation

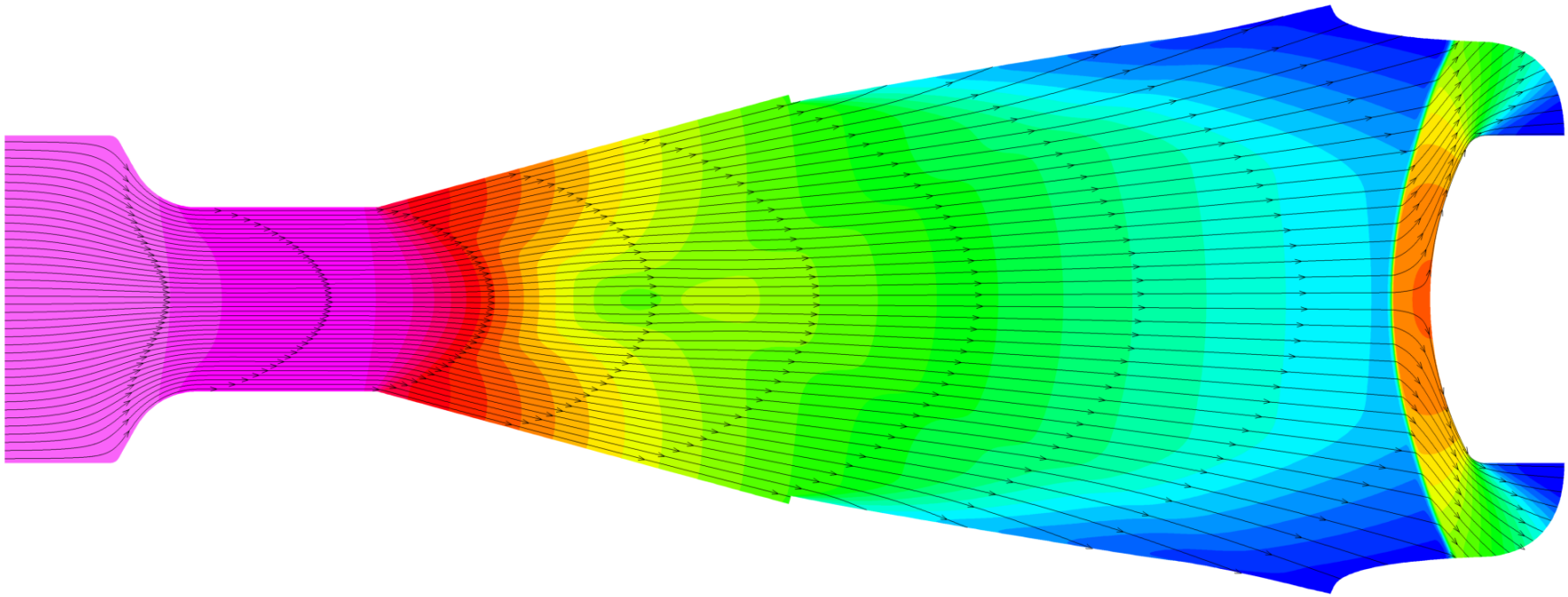


CFDLab

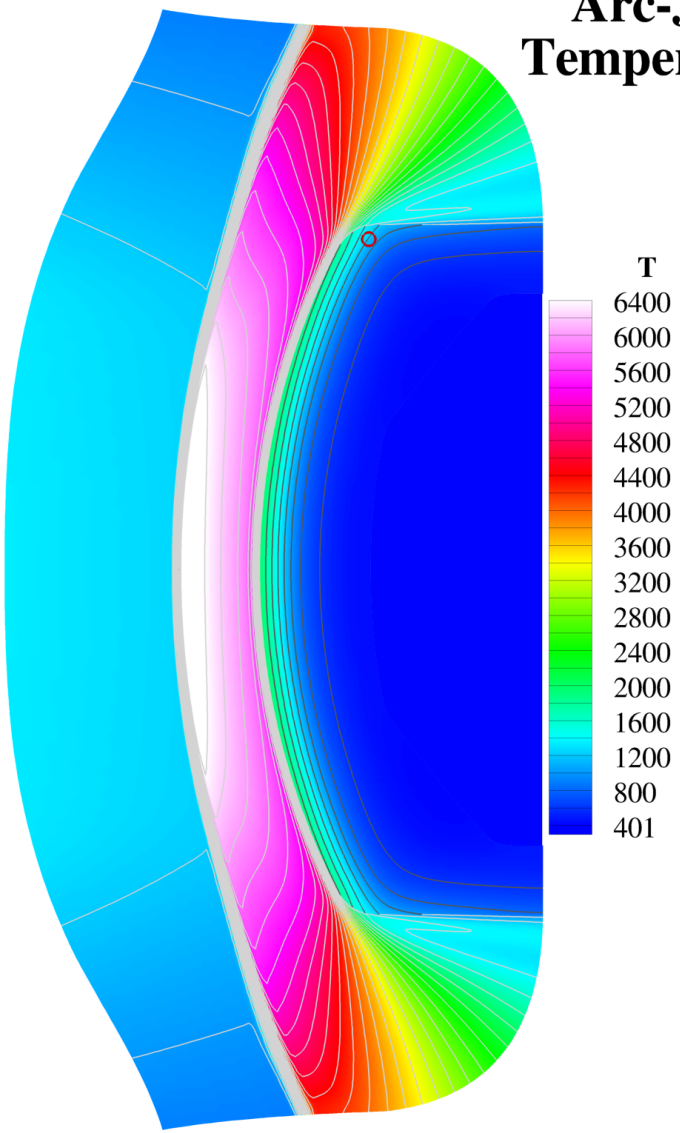
Arcjet Nozzle Calculation

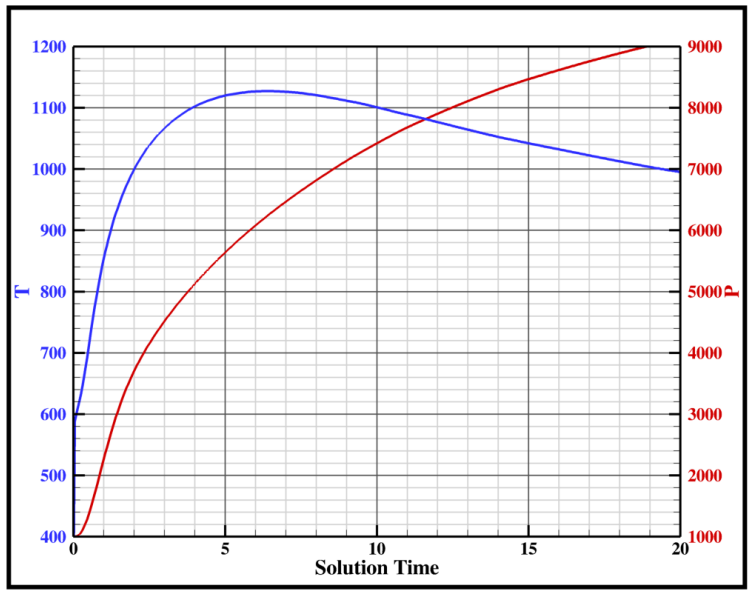


Arcjet Nozzle Calculation



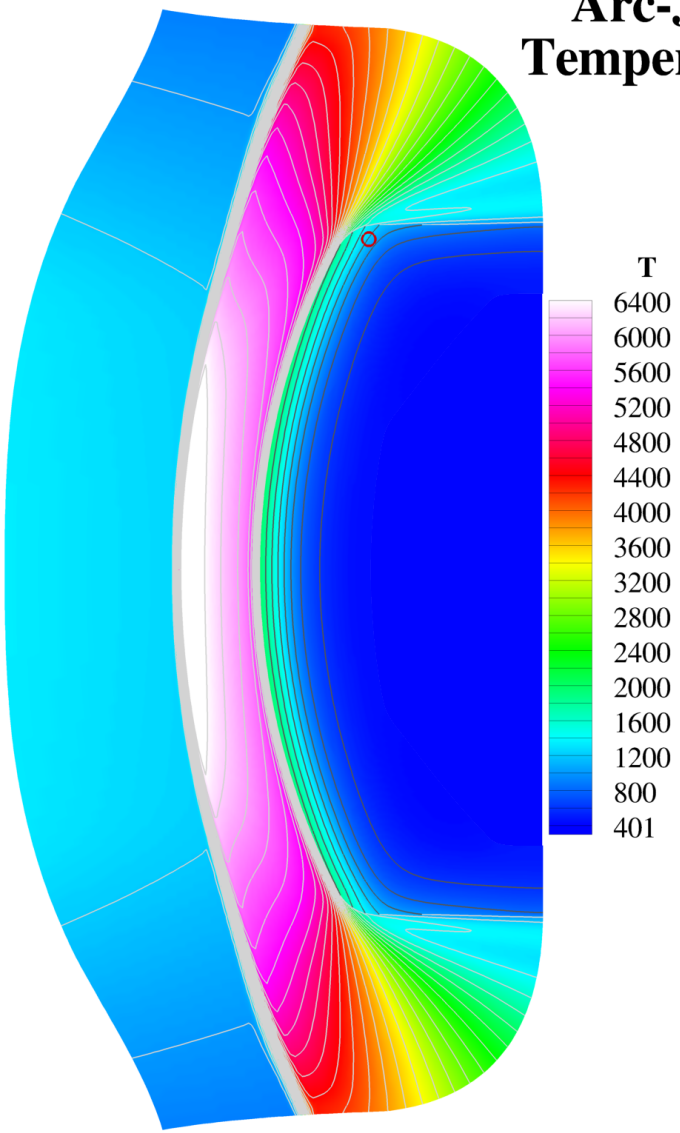
Coupled Pyrolysis, Temperature

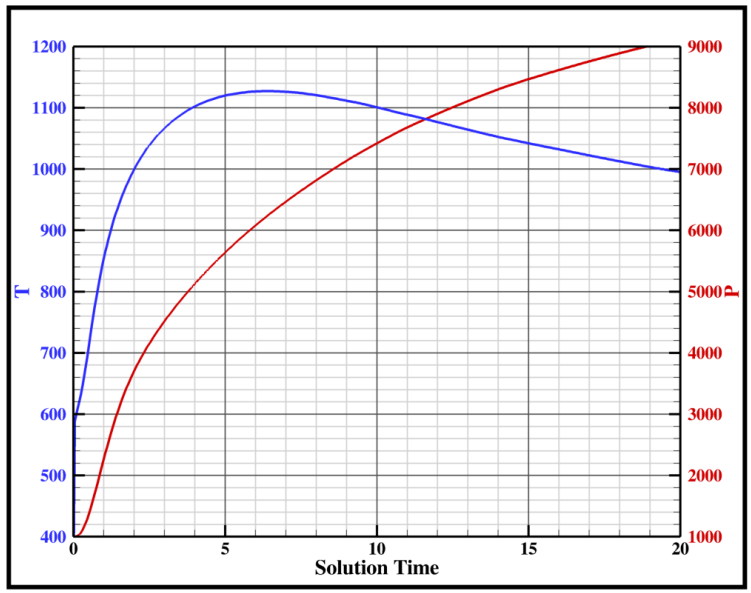




CFDLab

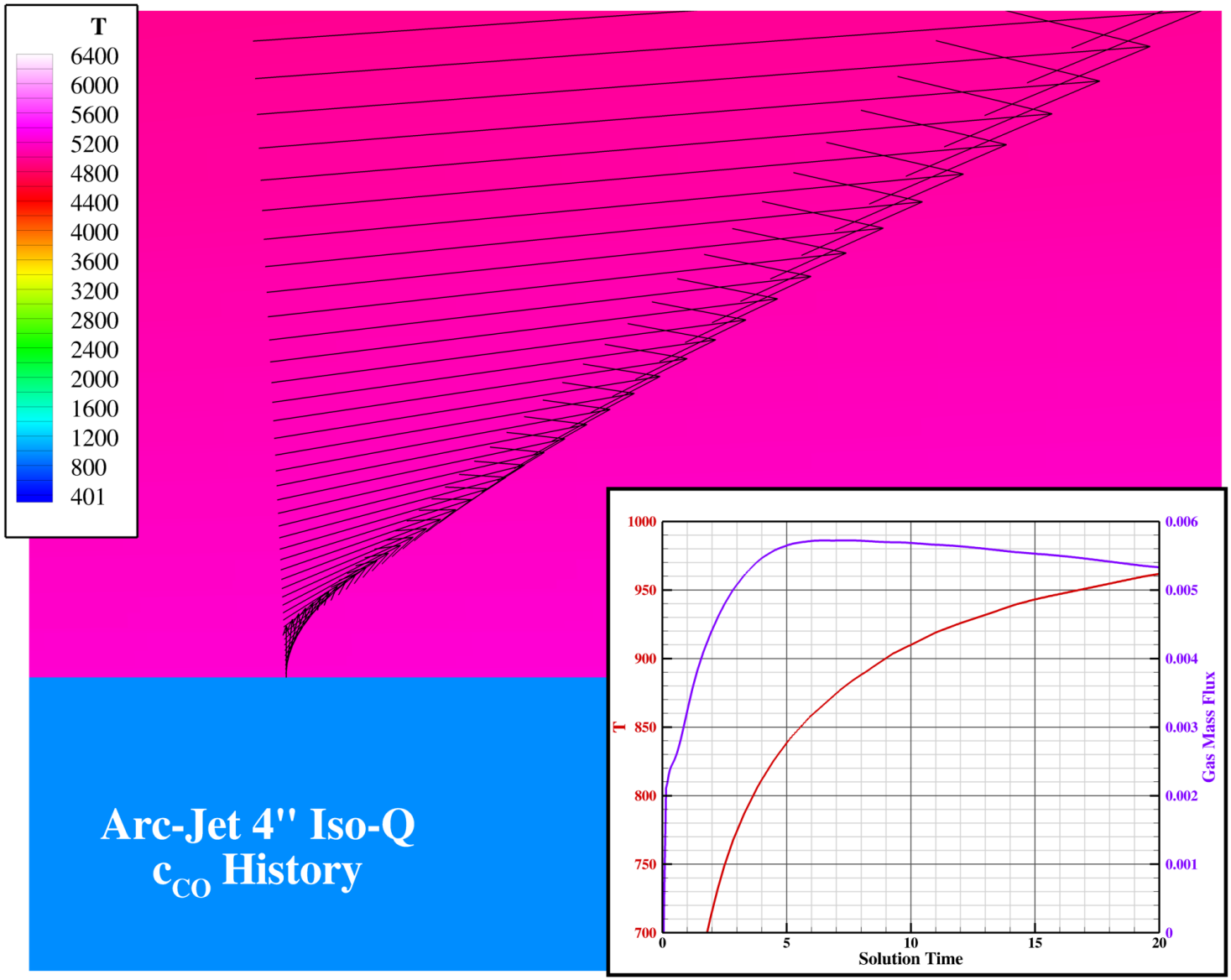
Coupled Pyrolysis, Temperature





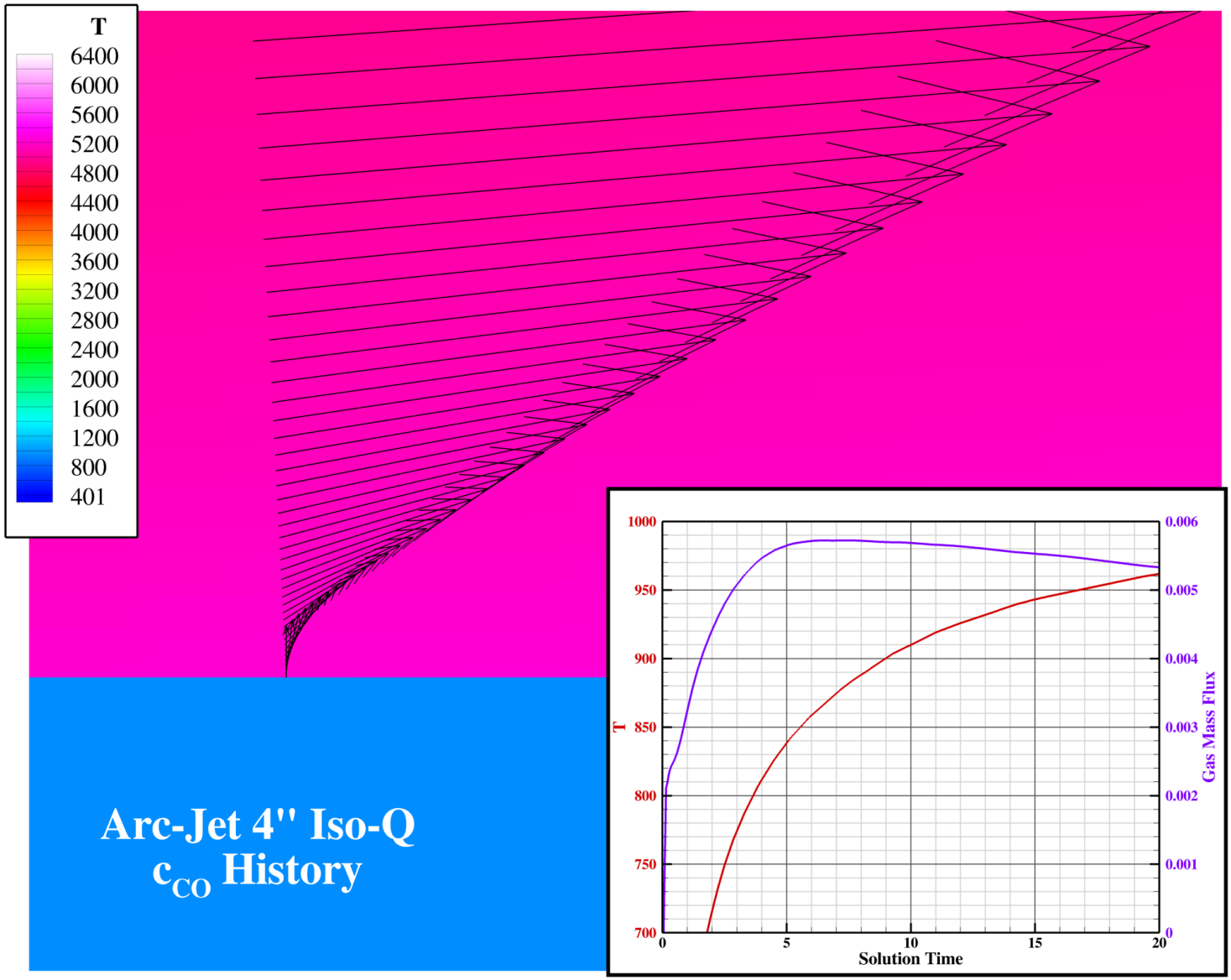
CFDLab

Coupled Pyrolysis, Pyrolysis gas mass flux, $\dot{m}$



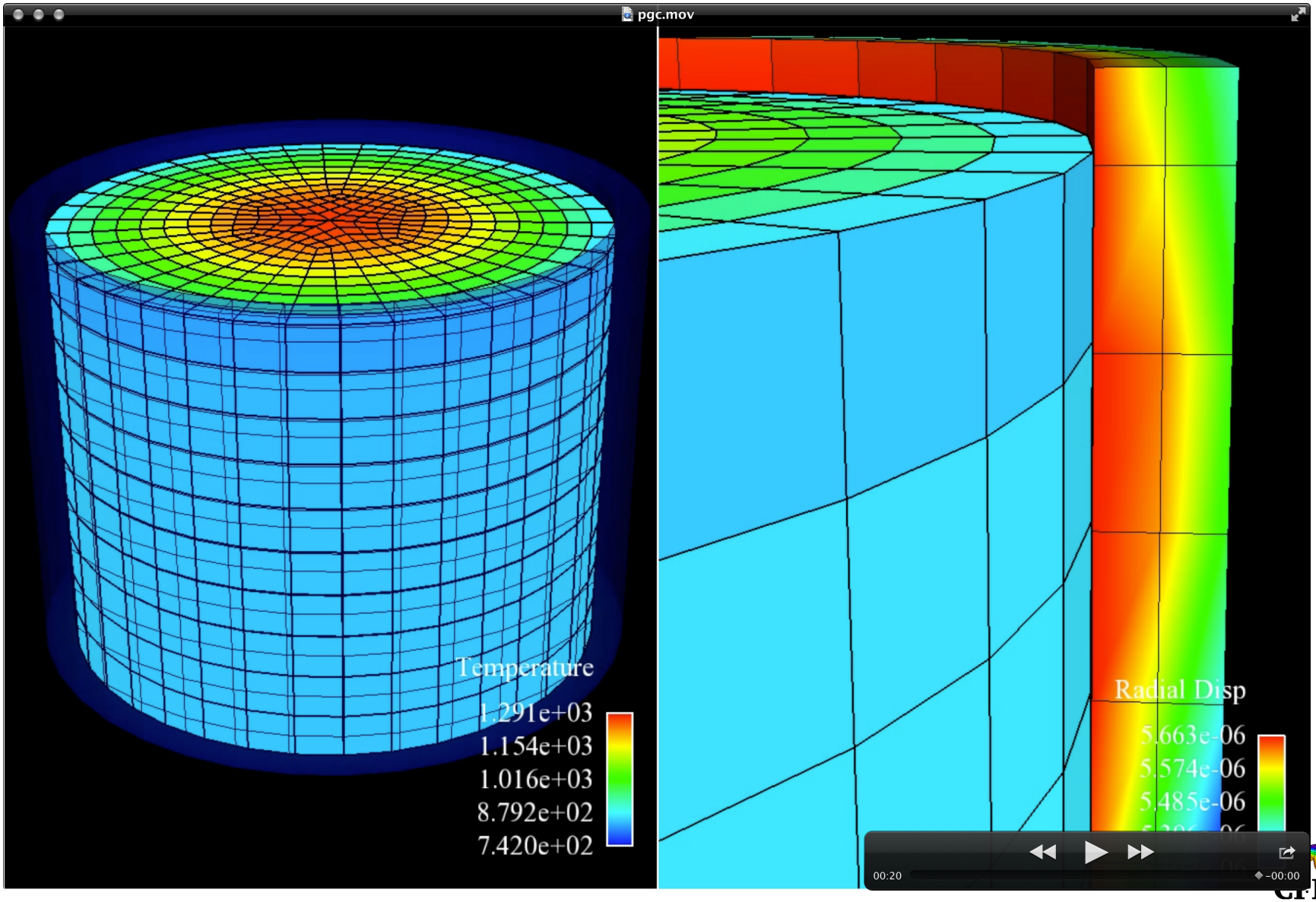
CFDLab

Coupled Pyrolysis, Pyrolysis gas mass flux, $\dot{m}$

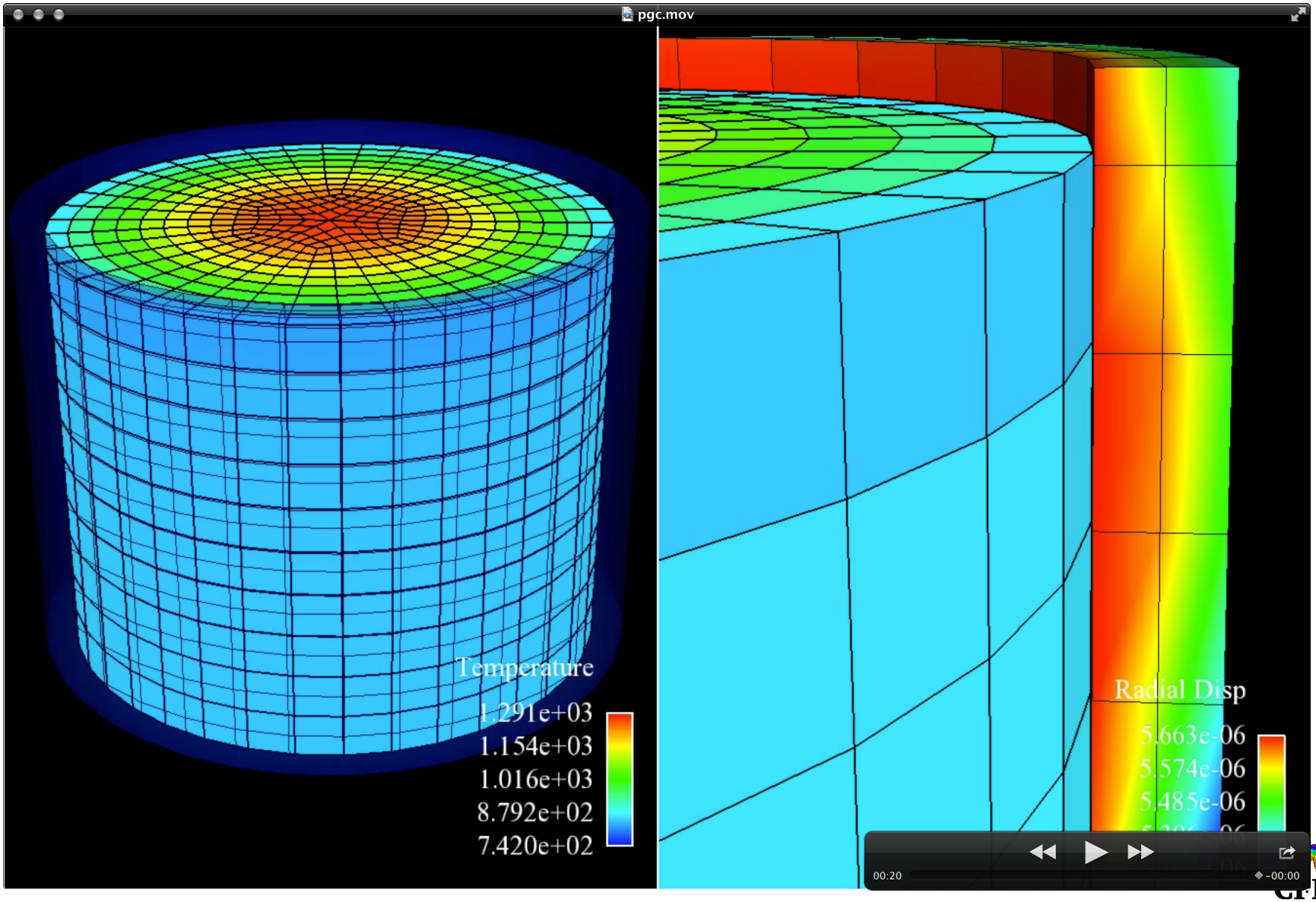


CFDLab

Coupled Thermal/Solid Mechanics



Coupled Thermal/Solid Mechanics



The MOOSE Framework - Gaston et al., INL

MOOSE

Derek Gaston, Idaho National Laboratory

Collaborators:
Dana Knoll, LANL
Glen Hansen, INL
Chris Newman, LANL
Ryosuke Park, INL
Cody Permann, INL
David Andrs, INL
Hai Huang, INL
Michael Tonks, INL
Robert Podgorney, INL

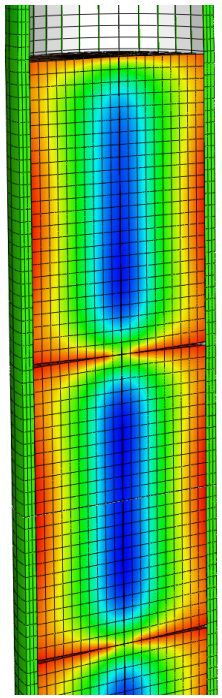
CFDLab

The MOOSE Framework - Gaston et al., INL

Idaho National Laboratory

MOOSE – Multiphysics Object Oriented Simulation Environment

- A framework for solving computational nuclear engineering problems in a well planned, managed, and coordinated way
  - *Leveraged across multiple programs*
- Designed to significantly reduce the expense and time required to develop new applications
- Designed to develop analysis tools
  - *Uses very robust solution methods*
  - *Designed to be easily extended and maintained*
  - *Efficient on both a few and many processors*
- Currently supports ~7 applications which are developed and used by ~20 scientists.



CFDLab

The MOOSE Framework - Gaston et al., INL

MOOSE Ecosystem

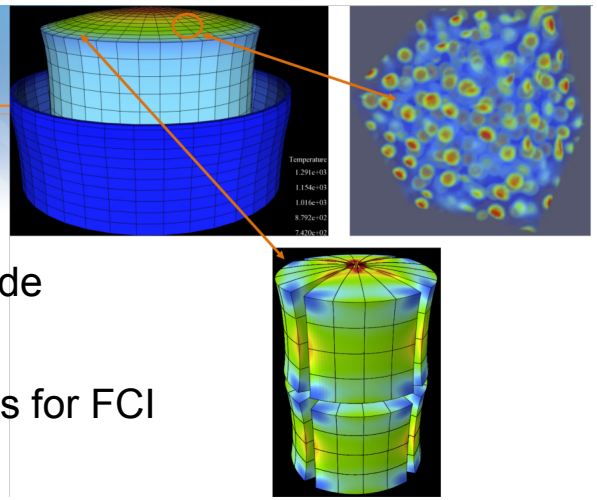
INL Idaho National Laboratory

Application	Physics	Start	Time To Results	Lines of Code
BISON	Thermo-mechanics, Chemical Diffusion, coupled mesoscale	June 2008	4 Months	931
PRONGHORN	Neutronics, Porous Flow, Eigenvalue	September 2008	3 Months	2,883
SALMON	Multiphase Porous Flow	June 2009	3 Months	800
MARMOT	4^{th} Order Phasefield Mesoscale	August 2009	1 Month	838
RAT	Porous ReActive Transport	August 2009	1 Month	439
FALCON	Geo-mechanics, coupled mesoscale	September 2009	3 Months	810

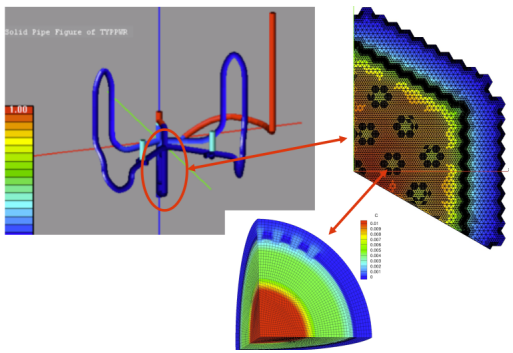
CFDLab

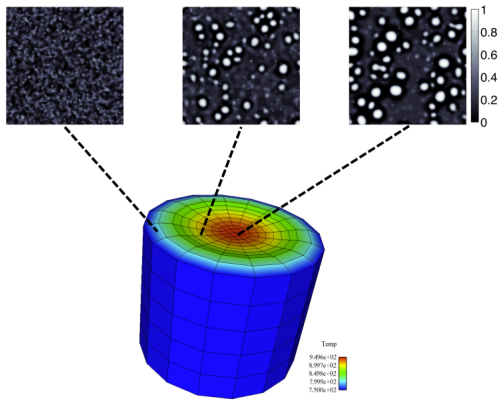
The MOOSE Framework - Gaston et al., INL

BISON fuel performance

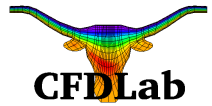


- LWR, Triso, and TRU fuel performance code
- Parallel 1D-3D thermomechanics code
- Thermal, mechanical, and chemical models for FCI
- Constituent redistribution
- Material, fission product swelling, fission gas release models
- Mesoscale-informed material models





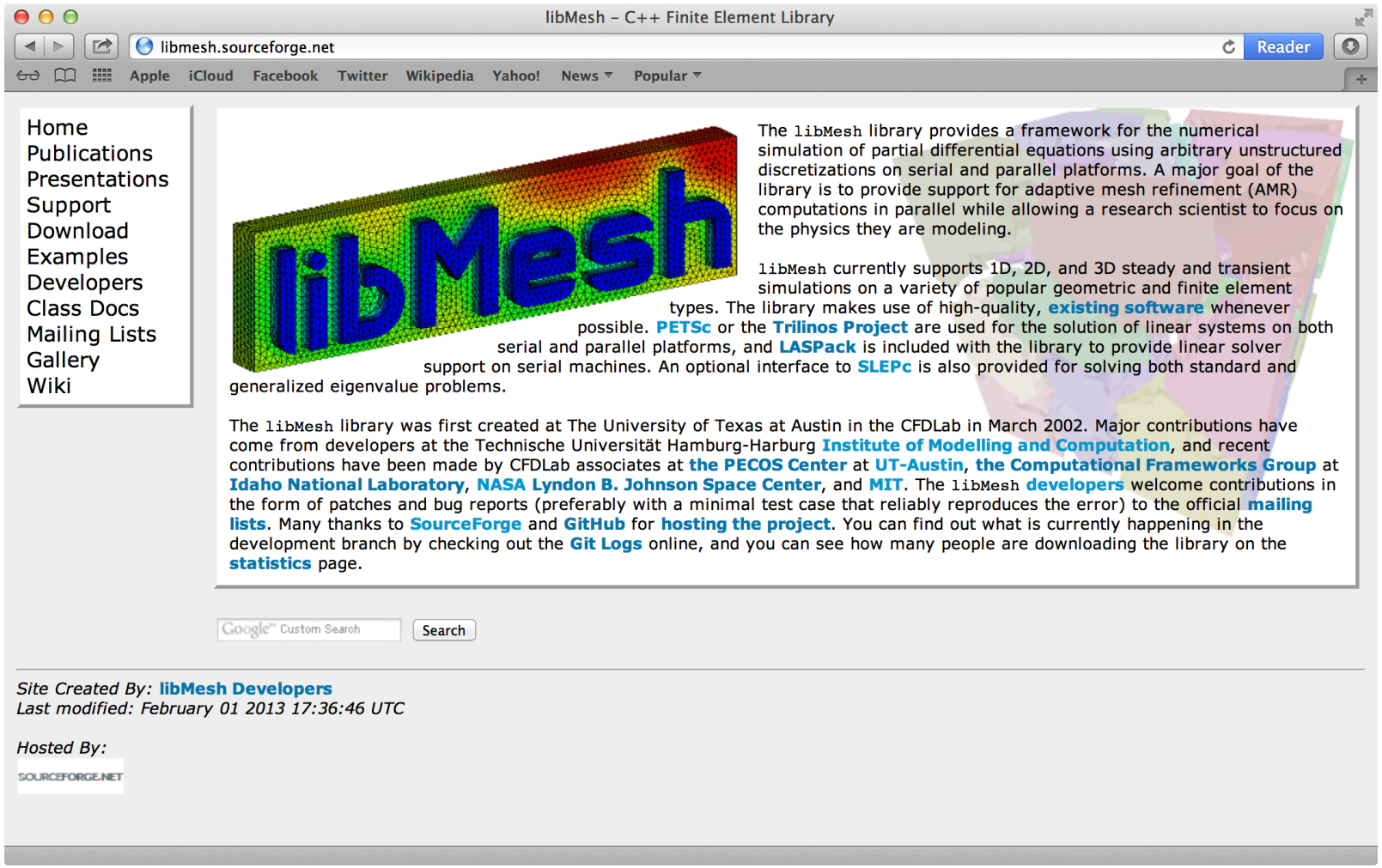
CFDLab



Important Websites

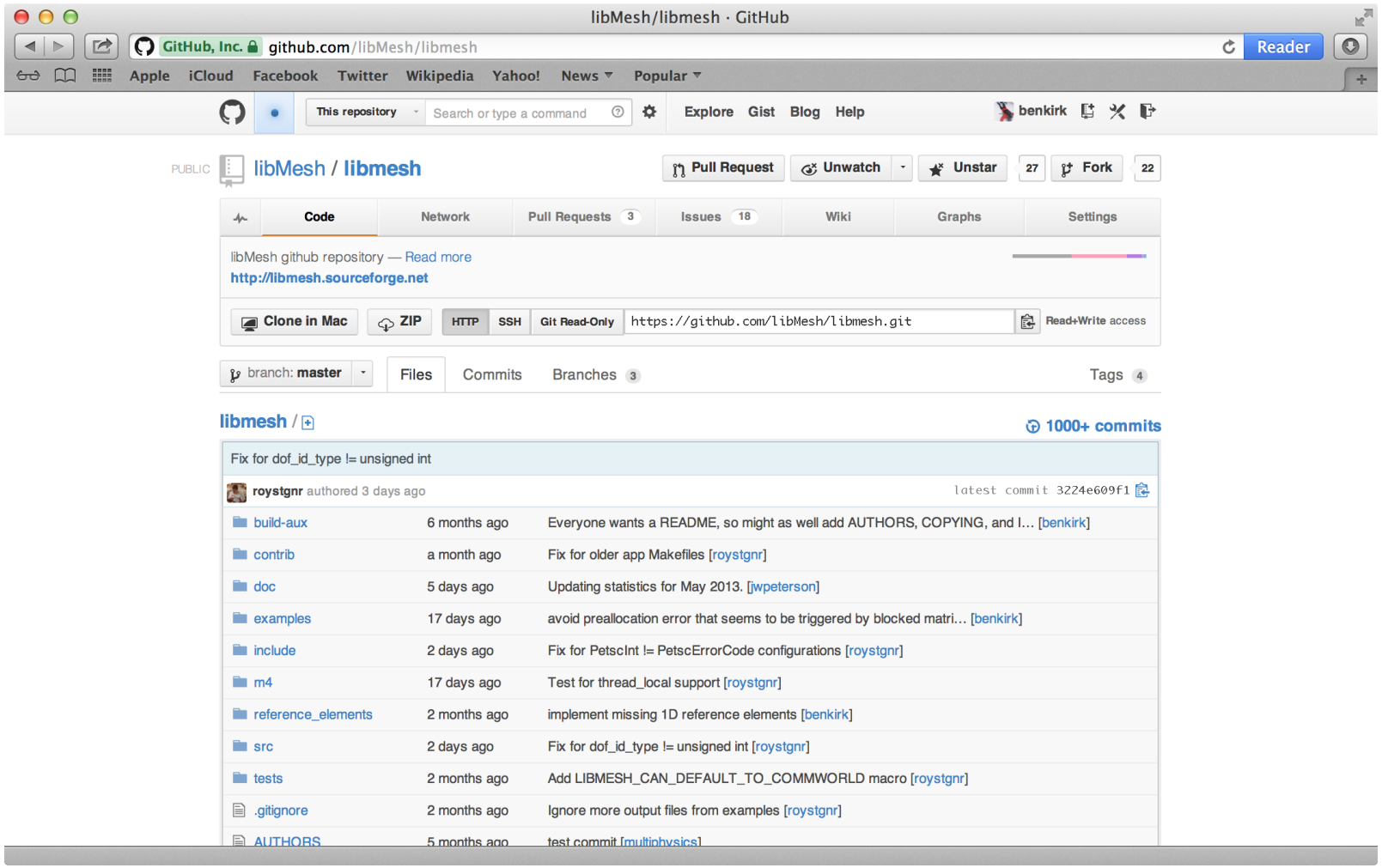
- Primary website
- Revision Control & Collaboration with GitHub
- Continuous Integration with Buildbot

http://libmesh.sourceforge.net

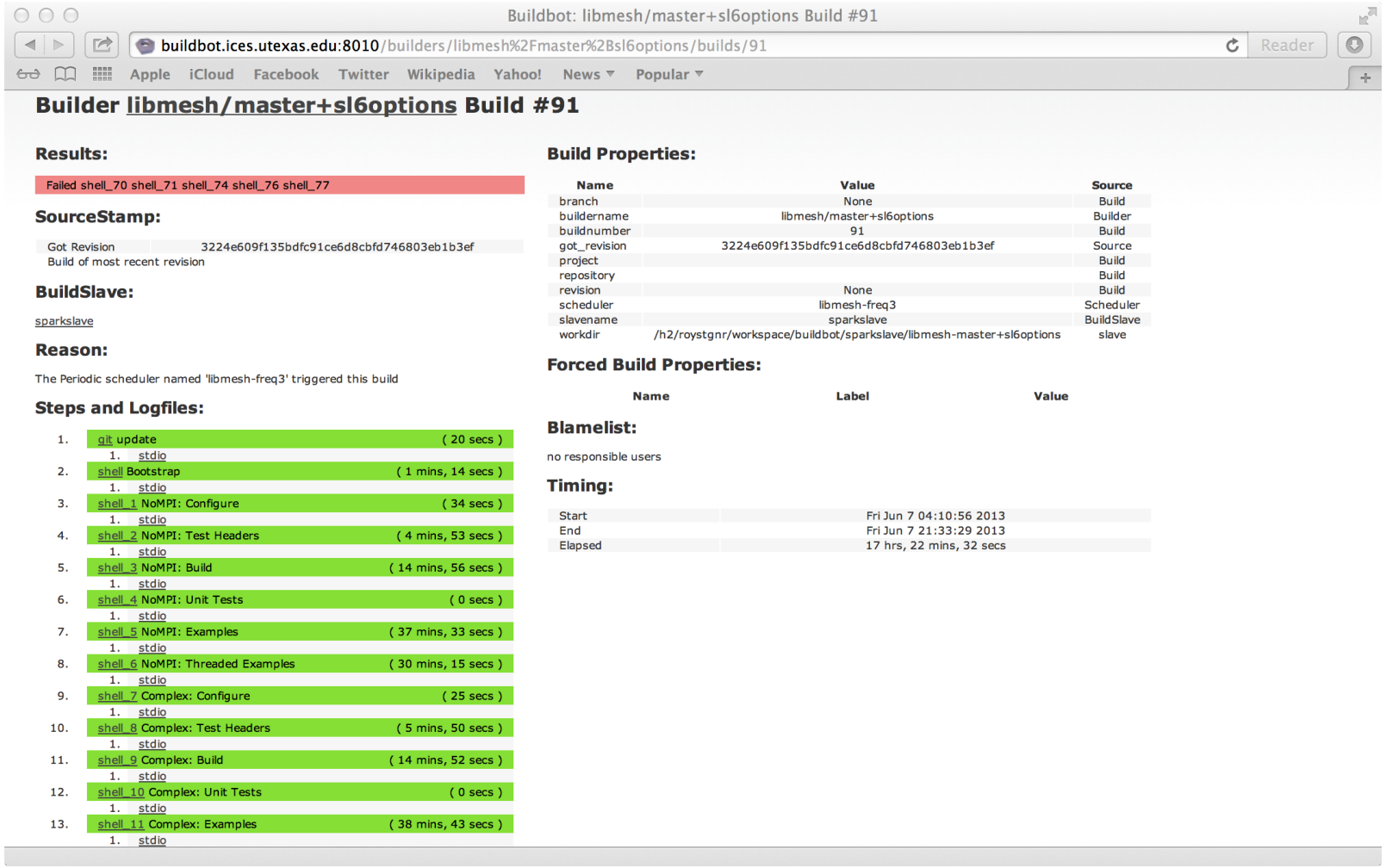


CFDLab

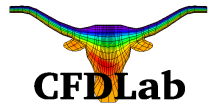
http://github.com/libMesh/libmesh



http://buildbot.ices.utexas.edu:8010/waterfall



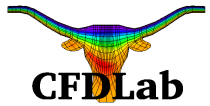
Building libMesh



Getting the libMesh Source

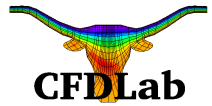
- **Blessed, Stable releases:**
  Download prepackaged releases from
  `http://sourceforge.net/projects/libmesh/files/libmesh`
- **Development tree:**
  Grab the latest source tree from GitHub:

```
$ git clone git://github.com/libMesh/libmesh.git
```



libMesh Suggested Dependencies

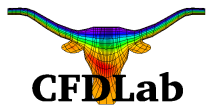
- MPI is of course required for shared-memory parallelism.
- Out of the box, libMesh will build with support for serial linear systems.
- Highly recommended you first install PETSc and/or Trilinos, which libMesh uses for solving linear systems in parallel.
- Other recommended, optional packages are:
  - SLEPc: eigenvalue support on top of PETSc.
  - Intel's Threading Building Blocks for shared-memory multithreading.



Building libMesh from source

Unpack, Configure, Build, Install, & Test

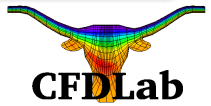
```
# unpack the distribution
$ tar jxf libmesh-0.9.1.tar.bz2 && cd libmesh-0.9.1
# configure, install into the current directory
$ ./configure --prefix=$PWD/install
# build & install
$ make -j 4 && make -j 4 install
# run all the examples, but only the optimized flavor
$ make -j 4 check METHODS=opt
```



Building libMesh from source

Advanced Configurations

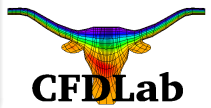
```
# unpack the distribution
$ tar jxf libmesh-0.9.1.tar.bz2 && cd libmesh-0.9.1
# build in a subdirectory, allows multiple simultaneous builds
$ mkdir -p clang && cd clang
# configure specifing optional packages & compilers
$ ../configure --prefix=$PWD/install \
               --with-glpk-include=/opt/local/include \
               --with-glpk-lib=/opt/local/lib \
               --with-vtk-include=/opt/local/include/vtk-5.10 \
               --with-vtk-lib=/opt/local/lib/vtk-5.10 \
               --with-eigen-include=/opt/local/include/eigen3 \
               --with-cxx=clang++-mp-3.3 --with-cc=clang-mp-3.3 \
               --disable-fortran
# build & install
$ make -j 4 && make -j 4 install
```



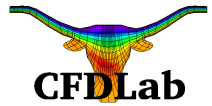
Building libMesh from source

Testing the Installation

```
$ make -j 4 installcheck
Making installcheck in include
Making installcheck in libmesh

Checking for standalone headers in installed tree ...

Testing Header libmesh/libmesh_config.h ...              [   OK   ]
Testing Header base/auto_ptr.h ...                       [   OK   ]
Testing Header base/dirichlet_boundaries.h ...           [   OK   ]
Testing Header base/dof_map.h ...                        [   OK   ]
Testing Header base/dof_object.h ...                     [   OK   ]
...

Checking for self-sufficient examples...

Testing examples in /tmp/libmesh-0.9.1/_inst/examples
Testing example installation adaptivity/ex1 ...          [   OK   ]
Testing example installation adaptivity/ex2 ...          [   OK   ]
Testing example installation adaptivity/ex3 ...          [   OK   ]
Testing example installation adaptivity/ex4 ...          [   OK   ]
Testing example installation adaptivity/ex5 ...          [   OK   ]
...
```



Getting Started with `libMesh` Applications



The libMesh installation

Installation Tree

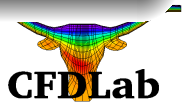
```
# henceforth we assume LIBMESH_DIR points to the installation path
$ cd $LIBMESH_DIR
$ ls
Make.common contrib     examples    lib
bin         etc         include     share
$ ls lib | grep mesh
libmesh_dbg.0.dylib
libmesh_dbg.dylib
libmesh_dbg.la
libmesh_devel.0.dylib
libmesh_devel.dylib
libmesh_devel.la
libmesh_opt.0.dylib
libmesh_opt.dylib
libmesh_opt.la
$ ls lib/pkgconfig
Make.common       libmesh-devel.pc libmesh-opt.pc   libmesh.pc
libmesh-dbg.pc    libmesh-oprof.pc libmesh-prof.pc  netcdf.pc
```

CFDLab

The libMesh installation

Compiling Simple Applications with pkg-config

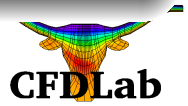
```
# make sure pkg-config can find the libMesh configuration
$ export PKG_CONFIG_PATH=$LIBMESH_DIR/lib/pkgconfig:$PKG_CONFIG_PATH

# copy the first example program
$ cp -r $LIBMESH_DIR/examples/introduction/ex1 . && cd ex1

# see what we've got
$ ls
Makefile introduction_ex1.C run.sh

# compile against the full debug version of libMesh
$ mpicxx -o introduction_ex1 introduction_ex1.C \
  `pkg-config --cflags --libs libmesh-dbg`

# compile against the default, optimized version of libMesh
$ mpicxx -o introduction_ex1 introduction_ex1.C \
  `pkg-config --cflags --libs libmesh`
```



The libMesh installation

Compiling Simple Applications with libmesh-config

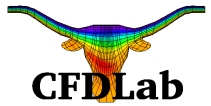
```
# we support a similar utility, libmesh-config, which predates
# pkg-config support. this is similar, but also can report the
# compiler used.
$ PATH=$LIBMESH_DIR/bin:$PATH

$ libmesh-config
usage: libmesh-config --cppflags --cxxflags --include --libs
       libmesh-config --cxx
       libmesh-config --cc
       libmesh-config --fc
       libmesh-config --fflags
       libmesh-config --version
       libmesh-config --host

# get the name of the compiler used, as passed to ./configure
$ libmesh-config --cxx
mpicxx
```



The libMesh installation

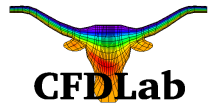
Compiling Applications with make

```
# copy the first example program
$ cp -r $LIBMESH_DIR/examples/introduction/ex1 . && cd ex1

# compile against the default, optimized version of libMesh
$ make
Compiling C++ (in optimized mode) introduction_ex1.C...
Linking example-opt...

# compile against the full debug version of libMesh
$ make METHOD=dbg
Compiling C++ (in debug mode) introduction_ex1.C...
Linking example-dbg...
```



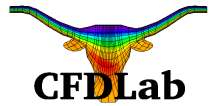
A first libMesh application

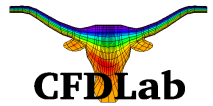
```
#include <iostream>
#include "libmesh/libmesh.h"
#include "libmesh/mesh.h"

using namespace libMesh;

int main (int argc, char** argv)
{
  LibMeshInit init (argc, argv);

  if (argc < 4)
    {
      if (libMesh::processor_id() == 0)
        std::cerr << "Usage: " << argv[0] << " -d 2 in.mesh [-o out.mesh]"
                  << std::endl;

      libmesh_error();
    }

  Mesh mesh(init.comm());

  mesh.read (argv[3]);

  mesh.print_info();

  if (argc >= 6 && std::string("-o") == argv[4])
    mesh.write (argv[5]);

  return 0;
}
```



A first libMesh application

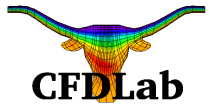
```
# copy & build the example
$ cp -r $LIBMESH_TUTORIAL/basic .
$ cd basic
$ make
Compiling C++ (in optimized mode) introduction_ex1.C...
Linking example-opt...

# run the example, reading a trivial mesh and writing output
$ ./example-opt -d 3 \
  $LIBMESH_DIR/share/reference_elements/3D/one_hex27.xda -o out.exo
 Mesh Information:
  mesh_dimension()=3
  spatial_dimension()=3
  n_nodes()=27
    n_local_nodes()=27
  n_elem()=1
    n_local_elem()=1
    n_active_elem()=1
  n_subdomains()=1
  n_partitions()=1
  n_processors()=1
  n_threads()=1
  processor_id()=0
```





Solving Problems the **libMesh** way

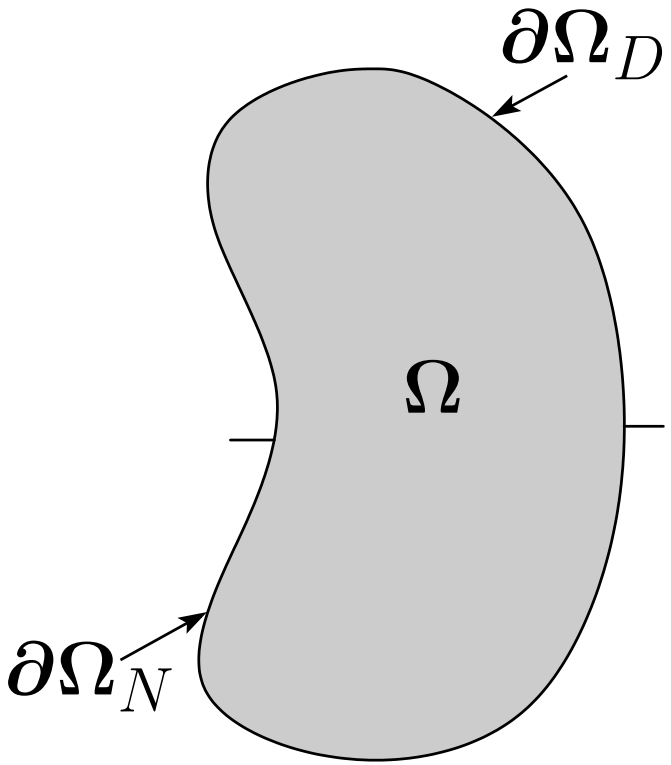
Discretizing a Generic Boundary Value Problem



A general class of PDE

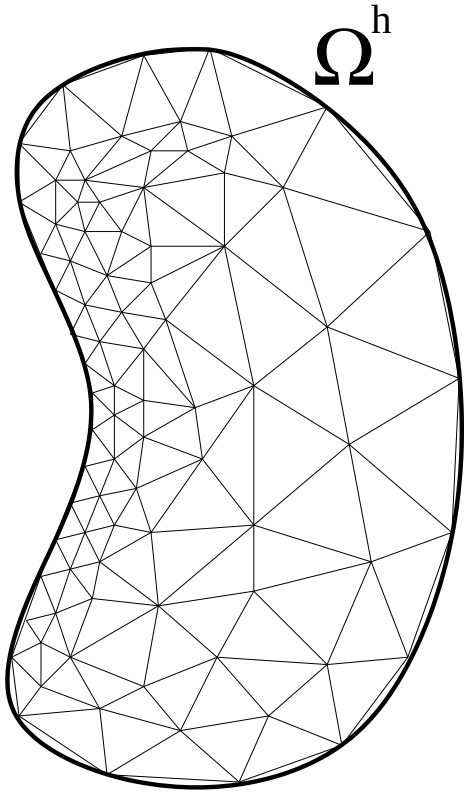
- We assume there is a Boundary Value Problem of the form to be approximated in an FE function space

$$
\begin{aligned}
M\frac{\partial u}{\partial t} &= F(u) && \in \Omega \\
G(u) &= 0 && \in \Omega \\
u &= u_D && \in \partial\Omega_D \\
N(u) &= 0 && \in \partial\Omega_N \\
u(\boldsymbol{x},0) &= u_0(\boldsymbol{x})
\end{aligned}
$$



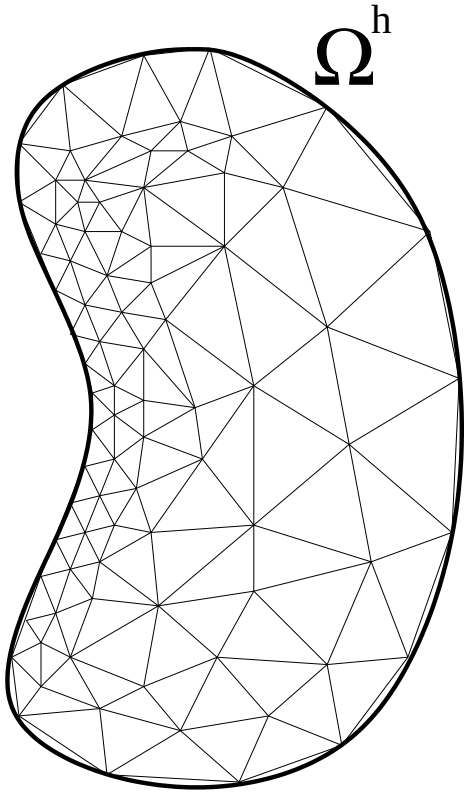
- Associated to the problem domain Ω is a `libMesh` data structure called a `Mesh`
- A `Mesh` is essentially a collection of finite elements

$$\Omega^h := \bigcup_e \Omega_e$$

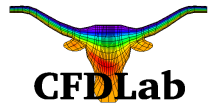


- Associated to the problem domain Ω is a `libMesh` data structure called a `Mesh`
- A `Mesh` is essentially a collection of finite elements

$$\Omega^h := \bigcup_e \Omega_e$$



- `libMesh` provides some simple structured mesh generation routines, interfaces to Triangle and TetGen, and supports a rich set of input file formats.



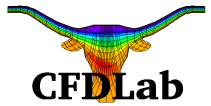
Data Types

`Real`	generally, a `double`, depending on `./configure`
`Number`	a `Real` or `std::complex<Real>`, depending on `./configure`
`Gradient`	a tuple of type `Number`, whose size is the spatial dimension

- `libMesh` can be compiled to support either real or complex-valued systems.

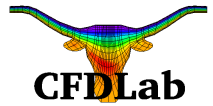
```
$ ./configure --enable-complex # turns on complex number support
```

- The underlying linear algebra libraries must support the requested type.



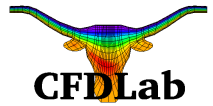
The Mesh

```
int main (int argc, char** argv)
{
  // Initialize the library.  This is necessary because the library
  // may depend on a number of other libraries (i.e. MPI and PETSc)
  // that require initialization before use.  When the LibMeshInit
  // object goes out of scope, other libraries and resources are
  // finalized.
  LibMeshInit init (argc, argv);

  // Create a mesh
  Mesh mesh;

  // Read the input mesh.
  mesh.read ("in.exo");

  // Print information about the mesh to the screen.
  mesh.print_info();

  // Write the output mesh
  mesh.write ("out.dat");

  ...
}
```



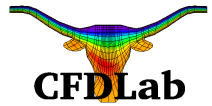
The Mesh

```
************************************************************
* Running Example introduction_ex1:
*   example-opt -d 3 reference_elements/3D/one_hex27.xda -o output.xda
************************************************************

 Mesh Information:
  mesh_dimension()=3
  spatial_dimension()=3
  n_nodes()=27
    n_local_nodes()=27
  n_elem()=1
    n_local_elem()=1
    n_active_elem()=1
  n_subdomains()=1
  n_partitions()=1
  n_processors()=1
  n_threads()=1
  processor_id()=0
```



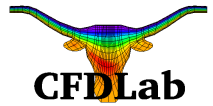
Operations on Objects in the `Mesh`

- From a `Mesh` it is trivial to access ranges of objects of interest through *iterators*.
- Iterators are simply a mechanism for accessing a range of objects.
- `libMesh` makes extensive use of *predictated iterators* to access, for example,
  - All elements in the mesh.
  - The "active" elements in the mesh assigned to the local processor in a parallel simulation.
  - The nodes in the mesh.



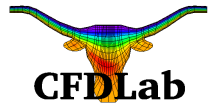
Mesh Iterators

```
void foo (const MeshBase &mesh)
{
  // Now we will loop over all the elements in the mesh that
  // live on the local processor. We will compute the element
  // matrix and right-hand-side contribution.  Since the mesh
  // may be refined we want to only consider the ACTIVE elements,
  // hence we use a variant of the \p active_elem_iterator.
  MeshBase::const_element_iterator
    el     = mesh.active_local_elements_begin();
  const MeshBase::const_element_iterator
    end_el = mesh.active_local_elements_end();

  for ( ; el != end_el; ++el)
    {
      // Store a pointer to the element we are currently
      // working on.  This allows for nicer syntax later.
      const Elem* elem = *el;
      ...
    }
  ...
}
```



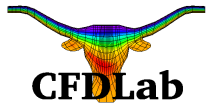
Mesh Iterators

```
void foo (const MeshBase &mesh)
{
  // We will now loop over all nodes.
  MeshBase::const_node_iterator       node_it  = mesh.nodes_begin();
  const MeshBase::const_node_iterator node_end = mesh.nodes_end();

  for ( ; node_it != node_end; ++node_it)
    {
      // the current node pointer
      const Node* node = *node_it;
      ...
    }
  ...
}
```



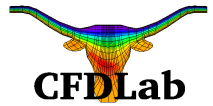
EquationSystems

- The `Mesh` is a discrete representation of the geometry for a problem.
- For a given `Mesh`, there can be an `EquationSystems` object, which represents one or more coupled system of equations posed on the `Mesh`.
  - There is only one `EquationSystems` object per `Mesh` object.
  - The `EquationSystems` object can hold many `System` objects, each representing a logical system of equations.
- High-level operations such as solution input/output is usually handled at the `EquationSystems` level.



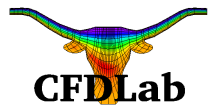
EquationSystems

```
...
// Create an equation systems object. This object can contain
// multiple systems of different flavors for solving loosely coupled
// physics.  Each system can contain multiple variables of different
// approximation orders. The EquationSystems object needs a
// reference to the mesh object, so the order of construction here
// is important.
EquationSystems equation_systems (mesh);

// Now we declare the system and its variables.  We begin by adding
// a "TransientLinearImplicitSystem" to the EquationSystems object,
// and we give it the name "Simple System".
equation_systems.add_system<TransientLinearImplicitSystem> ("Simple System");

// Adds the variable "u" to "Simple System".  "u" will be
// approximated using first-order approximation.
equation_systems.get_system("Simple System").add_variable("u", FIRST);

// Next we'll by add an "ExplicitSystem" to the EquationSystems
// object, and we give it the name "Complex System".
equation_systems.add_system<ExplicitSystem> ("Complex System");

// Give "Complex System" three variables -- each with a different
// approximation order.  Variables "c" and "T" will use first-order
// Lagrange approximation, while variable "dv" will use a
// second-order discontinuous approximation space.
equation_systems.get_system("Complex System").add_variable("c", FIRST);
equation_systems.get_system("Complex System").add_variable("T", FIRST);
equation_systems.get_system("Complex System").add_variable("dv", SECOND, MONOMIAL);

// Initialize the data structures for the equation system.
equation_systems.init();

// Prints information about the system to the screen.
equation_systems.print_info();
...
```



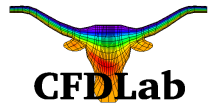
EquationSystems

```
EquationSystems
 n_systems()=2
  System #1, "Complex System"
   Type "Explicit"
   Variables={ "c" "T" } "dv"
   Finite Element Types="LAGRANGE" "MONOMIAL"
   Approximation Orders="FIRST" "SECOND"
   n_dofs()=222
   n_local_dofs()=60
   n_constrained_dofs()=0
   n_local_constrained_dofs()=0
   n_vectors()=1
   n_matrices()=0
  System #0, "Simple System"
   Type "TransientLinearImplicit"
   Variables="u"
   Finite Element Types="LAGRANGE"
   Approximation Orders="FIRST"
   n_dofs()=36
   n_local_dofs()=12
   n_constrained_dofs()=0
   n_local_constrained_dofs()=0
   n_vectors()=3
   n_matrices()=1
   DofMap Sparsity
     Average  On-Processor Bandwidth <= 6.22222
     Average Off-Processor Bandwidth <= 1.88889
     Maximum  On-Processor Bandwidth <= 10
     Maximum Off-Processor Bandwidth <= 8
```



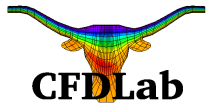
Elements

- The `Elem` base class defines a geometric element in `libMesh`.
- An `Elem` is defined by `Node`s, Edges (2D,3D) and Faces (3D).
- An `Elem` is sufficiently rich that in many cases it is the only argument required to provide to a function.



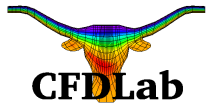
Elements

```
// access each Node on the element
for (unsigned int n=0; n<elem->n_nodes(); n++)
  {
    const Node *node = elem->get_node(n);

    // get a user-specified material property, based on
    // the subdomain the element belongs to
    const Real k_diff = my_matprop_func (elem->subdomain_id(), *node);
    ...
  }


// Perform some operation for elements on the boundary
for (unsigned int side=0; side<elem->n_sides(); side++)
  {
    // Every element knows its neigbor.  If it has no neighbor,
    // then it lies on a physical boundary.
    if (elem->neighbor(side) == NULL)
      {
        // Construct the side as a lower dimensional element
        AutoPtr<Elem> elem_side (elem->build_side(side));
        ...
      }
    ...
  }
```

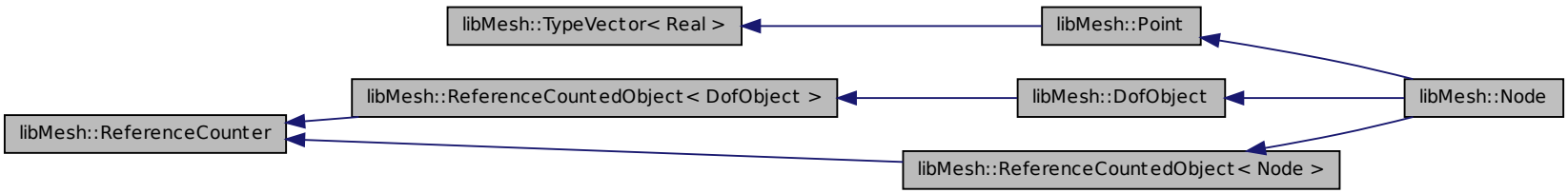


Nodes

- Nodes define spatial locations in arbitrary dimensions.
- Logically, a Node is a point in *N*-space plus metadata:
  - Global ID.
  - Processor ownership.
  - Degree of freedom indexing data.

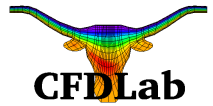


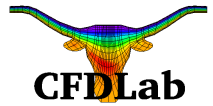
Nodes

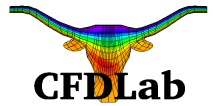

libMesh::TypeVector< Real >
libMesh::Point
libMesh::ReferenceCountedObject< DofObject >
libMesh::DofObject
libMesh::Node
libMesh::ReferenceCounter
libMesh::ReferenceCountedObject< Node >

Nodes

```
// loop over a range and determine the bounding box
void bounding_box(const ConstNodeRange &range)
{
  ...
  for (ConstNodeRange::const_iterator it = range.begin();
       it != range.end(); ++it)
    {
      const Node *node = *it;

      for (unsigned int i=0; i<LIBMESH_DIM; i++)
        {
          _vmin[i] = std::min(_vmin[i], (*node)(i));
          _vmax[i] = std::max(_vmax[i], (*node)(i));
        }
    }
  ...
}


...
// Query the number of DOFs for a particular node in a system
const unsigned int n_dofs_per_node = node->n_dofs(sys_num);
```







- The point of departure in any FE analysis which uses `libMesh` is the weighted residual statement

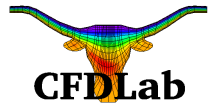
$$(F(u), v) = 0 \qquad \forall v \in \mathcal{V}$$

- The point of departure in any FE analysis which uses `libMesh` is the weighted residual statement

$$(F(u), v) = 0 \qquad \forall v \in \mathcal{V}$$

- Or, more precisely, the weighted residual statement associated with the finite-dimensional space $\mathcal{V}^h \subset \mathcal{V}$

$$(F(u^h), v^h) = 0 \qquad \forall v^h \in \mathcal{V}^h$$



Poisson Equation

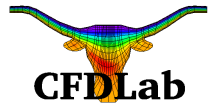
$$-\Delta u = f \qquad \in \quad \Omega$$

Poisson Equation

$$-\Delta u = f \qquad \in \quad \Omega$$

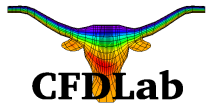
Weighted Residual Statement

$$(F(u), v) := \int_\Omega \left[\nabla u \cdot \nabla v - fv\right] dx + \int_{\partial\Omega_N} (\nabla u \cdot \boldsymbol{n})\, v \, ds$$



Linear Convection-Diffusion

$$-k\Delta u + \boldsymbol{b} \cdot \nabla u = f \qquad \in \quad \Omega$$



Linear Convection-Diffusion

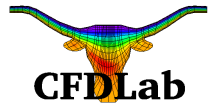
$$-k\Delta u + \boldsymbol{b} \cdot \nabla u = f \qquad \in \quad \Omega$$

Weighted Residual Statement

$$(F(u), v) := \int_{\Omega} \left[k\nabla u \cdot \nabla v + (\boldsymbol{b} \cdot \nabla u)v - fv \right] dx + \int_{\partial\Omega_N} k\left(\nabla u \cdot \boldsymbol{n}\right) v \, ds$$

Stokes Flow

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \Omega$$



Stokes Flow

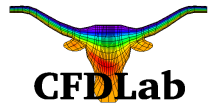
$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega$$

Weighted Residual Statement

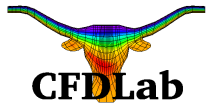
$$u := [\boldsymbol{u}, p] \quad , \quad v := [\boldsymbol{v}, q]$$

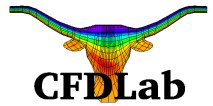
$$(F(u), v) := \int_{\Omega} \left[-p \left(\nabla \cdot \boldsymbol{v} \right) + \nu \nabla \boldsymbol{u} : \nabla \boldsymbol{v} - \boldsymbol{f} \cdot \boldsymbol{v} \right.$$

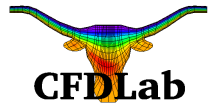
$$\left. + \left(\nabla \cdot \boldsymbol{u} \right) q \right] dx + \int_{\partial \Omega_N} \left(\nu \nabla \boldsymbol{u} - p \boldsymbol{I} \right) \boldsymbol{n} \cdot \boldsymbol{v} \, ds$$



- To obtain the approximate problem, we simply replace $u \leftarrow u^h$, $v \leftarrow v^h$, and $\Omega \leftarrow \Omega^h$ in the weighted residual statement.

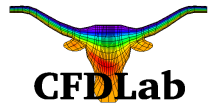






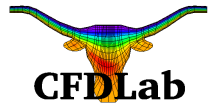
- For simplicity we start with the weighted residual statement arising from the Poisson equation, with $\partial\Omega_N = \emptyset$,

$$(F(u^h), v^h) := \int_{\Omega^h} \left[\nabla u^h \cdot \nabla v^h - f v^h\right] dx = 0 \qquad \forall v^h \in \mathcal{V}^h$$



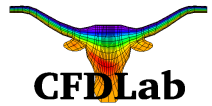
- The integral over Ω^h ...

$$0 = \int_{\Omega^h} \left[\nabla u^h \cdot \nabla v^h - f v^h \right] dx \qquad \forall v^h \in \mathcal{V}^h$$



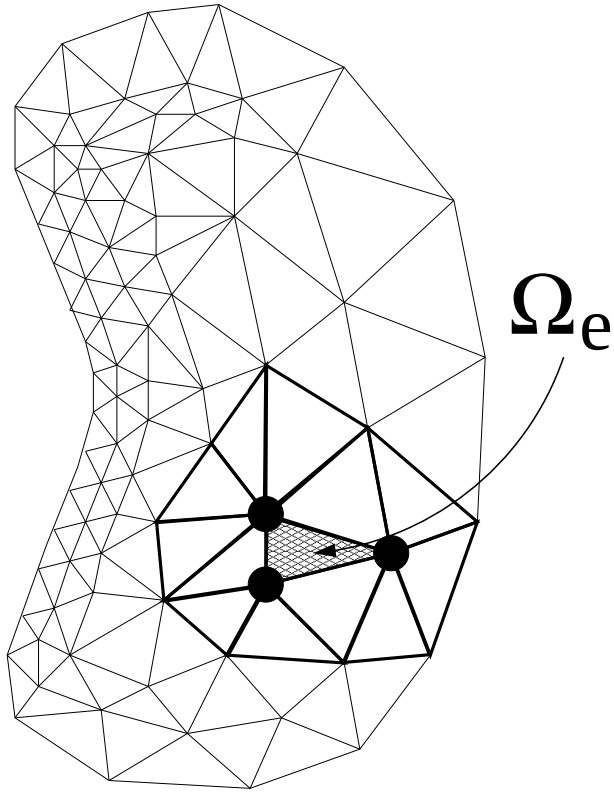
- The integral over Ω^h ... is written as a sum of integrals over the N_e finite elements:

$$
\begin{aligned}
0 &= \int_{\Omega^h} \left[\nabla u^h \cdot \nabla v^h - f v^h\right] dx \qquad \forall v^h \in \mathcal{V}^h \\
&= \sum_{e=1}^{N_e} \int_{\Omega_e} \left[\nabla u^h \cdot \nabla v^h - f v^h\right] dx \qquad \forall v^h \in \mathcal{V}^h
\end{aligned}
$$

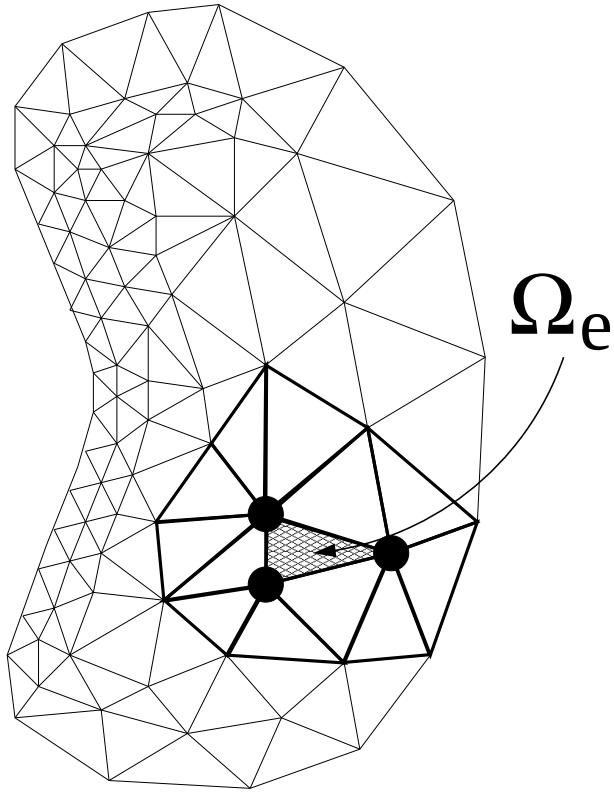


- An element integral will have contributions only from the global basis functions corresponding to its nodes.
- We call these local basis functions ϕ_i, $0 \leq i \leq N_s$.

$$v^h\big|_{\Omega_e} = \sum_{i=1}^{N_s} c_i \phi_i$$



- An element integral will have contributions only from the global basis functions corresponding to its nodes.
- We call these local basis functions ϕ_i, $0 \leq i \leq N_s$.



$$v^h\big|_{\Omega_e} = \sum_{i=1}^{N_s} c_i \phi_i$$

$$\int_{\Omega_e} v^h \, dx = \sum_{i=1}^{N_s} c_i \int_{\Omega_e} \phi_i \, dx$$

- The element integrals . . .

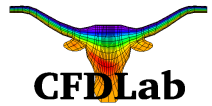
$$\int_{\Omega_e} \left[\nabla u^h \cdot \nabla v^h - f v^h \right] dx$$

- The element integrals . . .

$$\int_{\Omega_e} \left[\nabla u^h \cdot \nabla v^h - f v^h \right] dx$$

- are written in terms of the local "ϕ_i" basis functions

$$\sum_{j=1}^{N_s} u_j \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx - \int_{\Omega_e} f \phi_i \, dx \quad , \quad i = 1, \ldots, N_s$$



- The element integrals . . .

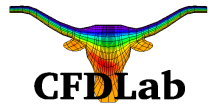
$$\int_{\Omega_e} \left[\nabla u^h \cdot \nabla v^h - f v^h \right] dx$$

- are written in terms of the local "ϕ_i" basis functions

$$\sum_{j=1}^{N_s} u_j \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx - \int_{\Omega_e} f \phi_i \, dx \quad , \quad i = 1, \ldots, N_s$$

- This can be expressed naturally in matrix notation as

$$\boldsymbol{K}^e \boldsymbol{U}^e - \boldsymbol{F}^e$$



- The entries of the element stiffness matrix are the integrals

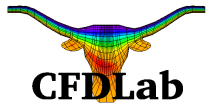
$$\boldsymbol{K}_{ij}^{e} := \int_{\Omega_e} \nabla\phi_j \cdot \nabla\phi_i \, dx$$

- The entries of the element stiffness matrix are the integrals

$$\boldsymbol{K}_{ij}^{e} := \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx$$

- While for the element right-hand side we have

$$\boldsymbol{F}_{i}^{e} := \int_{\Omega_e} f \phi_i \, dx$$



- The entries of the element stiffness matrix are the integrals

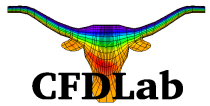
$$\boldsymbol{K}_{ij}^{e} := \int_{\Omega_e} \nabla \phi_j \cdot \nabla \phi_i \, dx$$

- While for the element right-hand side we have

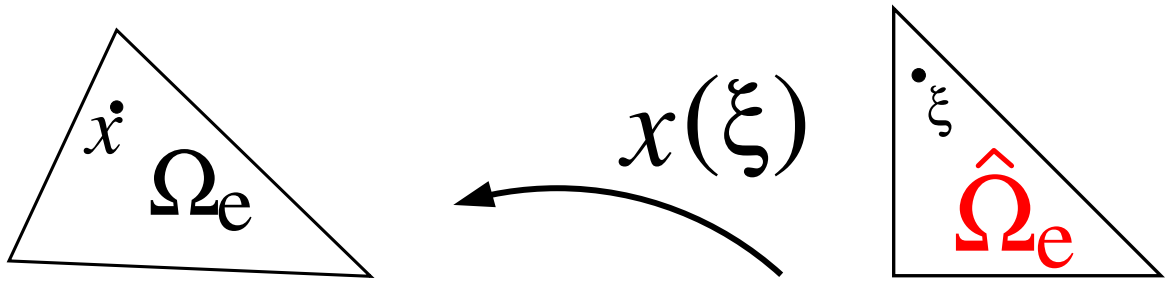
$$\boldsymbol{F}_{i}^{e} := \int_{\Omega_e} f \phi_i \, dx$$

- The element stiffness matrices and right-hand sides can be "assembled" to obtain the global system of equations

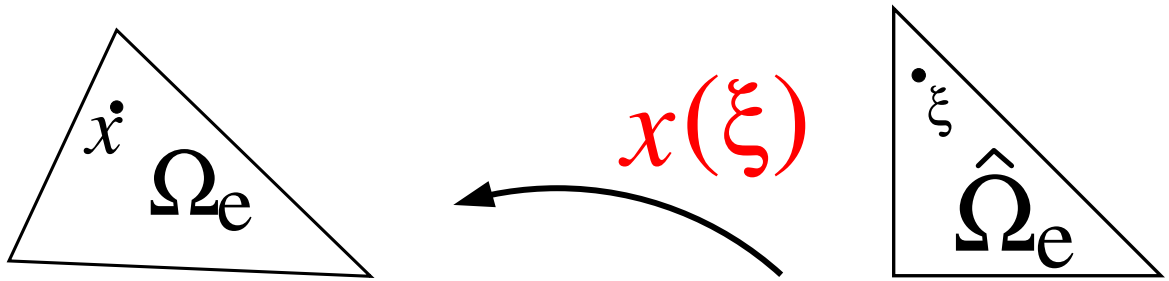
$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$



- The integrals are performed on a “reference” element $\hat{\Omega}_e$



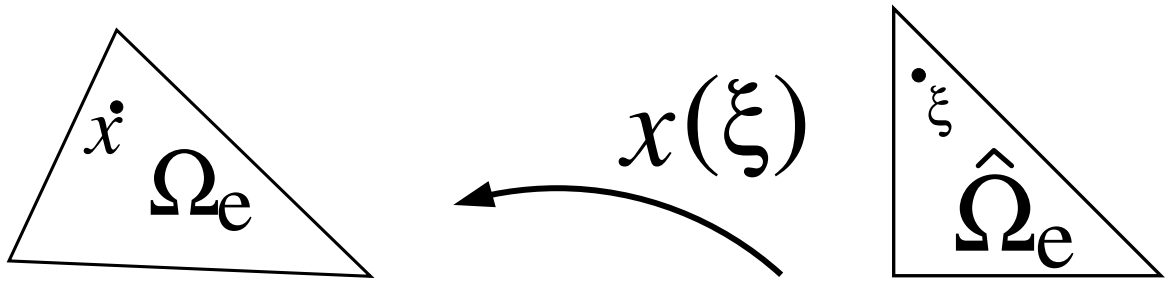
- The integrals are performed on a "reference" element $\hat{\Omega}_e$



- The Jacobian of the map $x(\xi)$ is J.

$$\boldsymbol{F}_i^e = \int_{\Omega_e} f\phi_i dx = \int_{\hat{\Omega}_e} f(x(\xi))\phi_i |\boldsymbol{J}| d\xi$$

- The integrals are performed on a "reference" element $\hat{\Omega}_e$



- Chain rule: $\nabla = J^{-1}\nabla_\xi := \hat{\nabla}_\xi$

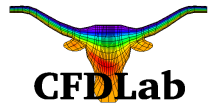
$$\boldsymbol{K}_{ij}^e = \int_{\Omega_e} \nabla\phi_j \cdot \nabla\phi_i \, dx = \int_{\hat{\Omega}_e} \hat{\nabla}_\xi\phi_j \cdot \hat{\nabla}_\xi\phi_i \, |J| d\xi$$

- The integrals on the “reference” element are approximated via numerical quadrature.

- The integrals on the "reference" element are approximated via numerical quadrature.
- The quadrature rule has N_q points "ξ_q" and weights "w_q".

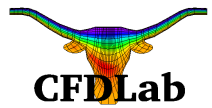
- The integrals on the "reference" element are approximated via numerical quadrature.
- The quadrature rule has N_q points "ξ_q" and weights "w_q".

$$\begin{aligned} \boldsymbol{F}_i^e &= \int_{\hat{\Omega}_e} f\phi_i |\boldsymbol{J}| d\xi \\ &\approx \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|\boldsymbol{J}(\xi_q)|w_q \end{aligned}$$



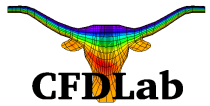
- The integrals on the "reference" element are approximated via numerical quadrature.
- The quadrature rule has N_q points "ξ_q" and weights "w_q".

$$\begin{aligned} \boldsymbol{K}_{ij}^{e} &= \int_{\hat{\Omega}_e} \hat{\nabla}_\xi \phi_j \cdot \hat{\nabla}_\xi \phi_i \, |J| d\xi \\ &\approx \sum_{q=1}^{N_q} \hat{\nabla}_\xi \phi_j(\xi_q) \cdot \hat{\nabla}_\xi \phi_i(\xi_q) |J(\xi_q)| w_q \end{aligned}$$



- libMesh provides the following variables at each quadrature point q

Code	Math	Description
JxW[q]	$\lvert J(\xi_q)\rvert w_q$	Jacobian times weight
phi[i][q]	$\phi_i(\xi_q)$	value of i^{th} shape fn.
dphi[i][q]	$\hat{\nabla}_\xi \phi_i(\xi_q)$	value of i^{th} shape fn. gradient
xyz[q]	$x(\xi_q)$	location of ξ_q in physical space



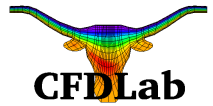
- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$\boldsymbol{F}_i^e = \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|J(\xi_q)|w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$\boldsymbol{F}_i^e = \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|J(\xi_q)|w_q$$

- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

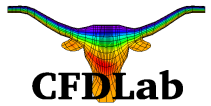
```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

$$\boldsymbol{F}_i^e = \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|J(\xi_q)|w_q$$

- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

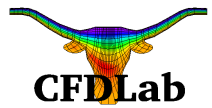
$$\boldsymbol{F}_i^e = \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|J(\xi_q)|w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

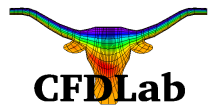
$$\boldsymbol{K}_{ij}^{e} = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi}\phi_j(\xi_q) \cdot \hat{\nabla}_{\xi}\phi_i(\xi_q)|\boldsymbol{J}(\xi_q)|w_q$$



- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

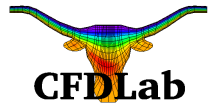
$$\boldsymbol{K}_{ij}^{e} = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi}\phi_j(\xi_q) \cdot \hat{\nabla}_{\xi}\phi_i(\xi_q)|\boldsymbol{J}(\xi_q)|w_q$$



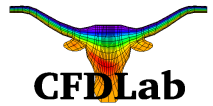
- The `libMesh` representation of the matrix and rhs assembly is similar to the mathematical statements.

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(dphi[j][q]*dphi[i][q]);
  }
```

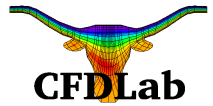
$$\boldsymbol{K}_{ij}^{e} = \sum_{q=1}^{N_q} \hat{\nabla}_{\xi}\phi_j(\xi_q) \cdot \hat{\nabla}_{\xi}\phi_i(\xi_q)|\boldsymbol{J}(\xi_q)|w_q$$



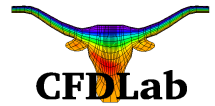
```
// We now define the matrix assembly function for the Poisson system
// by computing the element matrices and right-hand sides.  We are
// omitting BCs for brevity.
void assemble_poisson(EquationSystems& es,
                      const std::string& system_name)
{
  // Get a constant reference to the mesh object.
  const MeshBase& mesh = es.get_mesh();

  // The dimension that we are running
  const unsigned int dim = mesh.mesh_dimension();

  // Get a reference to the LinearImplicitSystem we are solving
  LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Poisson");

  // A reference to the DofMap object for this system.  The DofMap
  // object handles the index translation from node and element
  // numbers to degree of freedom numbers.  We will talk more about
  // the DofMap in future examples.
  const DofMap& dof_map = system.get_dof_map();

  // Get a constant reference to the Finite Element type for the first
  // (and only) variable in the system.
  FEType fe_type = dof_map.variable_type(0);

  // Build a Finite Element object of the specified type.  Since the
  // FEBase::build() member dynamically creates memory we will store
  // the object as an AutoPtr<FEBase>.  This can be thought of as a
  // pointer that will clean up after itself.
  AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));

  // A 5th order Gauss quadrature rule for numerical integration.
  QGauss qrule (dim, FIFTH);
```



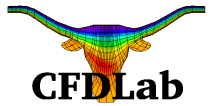
```
// Tell the finite element object to use our quadrature rule.
fe->attach_quadrature_rule (&qrule);

// Here we define some references to cell-specific data that will be
// used to assemble the linear system.  We begin with the element
// Jacobian * quadrature weight at each integration point.
const std::vector<Real>& JxW = fe->get_JxW();

// The physical XY locations of the quadrature points on the
// element.  These might be useful for evaluating spatially varying
// material properties at the quadrature points.
const std::vector<Point>& q_point = fe->get_xyz();

// The element shape functions evaluated at the quadrature points.
const std::vector<std::vector<Real> >& phi = fe->get_phi();

// The element shape function gradients evaluated at the quadrature
// points.
const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();

// Define data structures to contain the element matrix and
// right-hand-side vector contribution.  Following basic finite
// element terminology we will denote these "Ke" and "Fe".
DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

// This vector will hold the degree of freedom indices for the
// element.  These define where in the global system the element
// degrees of freedom get mapped.
std::vector<dof_id_type> dof_indices;

// Now we will loop over all the elements in the mesh.  We will
```



```
// compute the element matrix and right-hand-side contribution.  See
// example 3 for a discussion of the element iterators.
MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();

for ( ; el != end_el; ++el)
  {
    // Store a pointer to the element we are currently working on.
    // This allows for nicer syntax later.
    const Elem* elem = *el;

    // Get the degree of freedom indices for the current element.
    // These define where in the global matrix and right-hand-side
    // this element will contribute to.
    dof_map.dof_indices (elem, dof_indices);

    // Compute the element-specific data for the current element.
    // This involves computing the location of the quadrature points
    // (q_point) and the shape functions (phi, dphi) for the current
    // element.
    fe->reinit (elem);

    // Zero the element matrix and right-hand side before summing
    // them.  We use the resize member here because the number of
    // degrees of freedom might have changed from the last element.
    // Note that this will be the case if the element type is
    // different (i.e. the last element was a triangle, now we are
    // on a quadrilateral).
    Ke.resize (dof_indices.size(),
               dof_indices.size());

    Fe.resize (dof_indices.size());
```

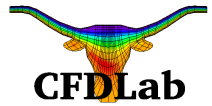


```
      // Now we will build the element matrix.  This involves a double
      // loop to integrate the test funcions (i) against the trial
      // functions (j).
      for (unsigned int qp=0; qp<qrule.n_points(); qp++)
        for (unsigned int i=0; i<phi.size(); i++)
          for (unsigned int j=0; j<phi.size(); j++)
            Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);

      // Now we build the element right-hand-side contribution.  This
      // involves a single loop in which we integrate the "forcing
      // function" in the PDE against the test functions.
      for (unsigned int qp=0; qp<qrule.n_points(); qp++)
        for (unsigned int i=0; i<phi.size(); i++)
          Fe(i) += JxW[qp]*10.*phi[i][qp];

      // If we are using an adaptive mesh this will apply any hanging
      // node constraint equations
      dof_map.heterogenously_constrain_element_matrix_and_vector (Ke, Fe, dof_indices);

      // The element matrix and right-hand-side are now built for this
      // element.  Add them to the global matrix and right-hand-side
      // vector.  The SparseMatrix::add_matrix() and
      // NumericVector::add_vector() members do this for us.
      system.matrix->add_matrix (Ke, dof_indices);
      system.rhs->add_vector    (Fe, dof_indices);
    }
}
```



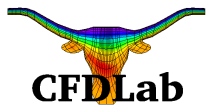
A Complete Program:

`poisson`



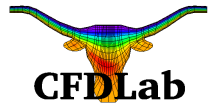
Poisson class definition

```
// headers omitted for brevity
class Poisson : public System::Assembly
{
public:
  Poisson (EquationSystems &es_in) :
    es (es_in)
  {}

  void assemble ();

  Real exact_solution (const Real x,
                       const Real y,
                       const Real z = 0.) const
  {
    static const Real pi = acos(-1.);

    return cos(.5*pi*x)*sin(.5*pi*y)*cos(.5*pi*z);
  }

private:
  EquationSystems &es;
};
```



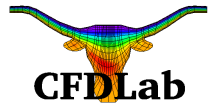
Poisson `main()` I

```
// C++ include files that we need
#include <iostream>
#include <algorithm>
#include <math.h>
#include <set>

// Basic include file needed for the mesh functionality.
#include "libmesh/libmesh.h"
#include "libmesh/mesh.h"
#include "libmesh/mesh_generation.h"
#include "libmesh/exodusII_io.h"
#include "libmesh/gnuplot_io.h"
#include "libmesh/linear_implicit_system.h"
#include "libmesh/equation_systems.h"

// Define the Finite Element object.
#include "libmesh/fe.h"

// Define Gauss quadrature rules.
#include "libmesh/quadrature_gauss.h"

// Define the DofMap, which handles degree of freedom
// indexing.
#include "libmesh/dof_map.h"

// Define useful datatypes for finite element
// matrix and vector components.
#include "libmesh/sparse_matrix.h"
#include "libmesh/numeric_vector.h"
#include "libmesh/dense_matrix.h"
```



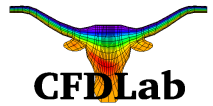
Poisson `main()` II

```
#include "libmesh/dense_vector.h"

// Define the PerfLog, a performance logging utility.
// It is useful for timing events in a code and giving
// you an idea where bottlenecks lie.
#include "libmesh/perf_log.h"

// The definition of a geometric element
#include "libmesh/elem.h"

// To impose Dirichlet boundary conditions
#include "libmesh/dirichlet_boundaries.h"
#include "libmesh/analytic_function.h"

#include "libmesh/string_to_enum.h"
#include "libmesh/getpot.h"

#include "poisson_problem.h"

// Bring in everything from the libMesh namespace
using namespace libMesh;



// Exact solution function prototype.
Real exact_solution (const Real x,
                     const Real y,
                     const Real z = 0.);

// Define a wrapper for exact_solution that will be needed below
```



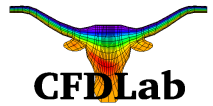
Poisson `main()` III

```
void exact_solution_wrapper (DenseVector<Number>& output,
                             const Point& p,
                             const Real)
{
  output(0) = exact_solution(p(0),
                             (LIBMESH_DIM>1)?p(1):0,
                             (LIBMESH_DIM>2)?p(2):0);
}

// Begin the main program.
int main (int argc, char** argv)
{
  // Initialize libMesh and any dependent libaries, like in example 2.
  LibMeshInit init (argc, argv);

  // Declare a performance log for the main program
  // PerfLog perf_main("Main Program");

  // Create a GetPot object to parse the command line
  GetPot command_line (argc, argv);

  // Check for proper calling arguments.
  if (argc < 3)
    {
      if (libMesh::processor_id() == 0)
        std::cerr << "Usage:\n"
                  <<"\t " << argv[0] << " -d 2(3)" << " -n 15"
                  << std::endl;

      // This handy function will print the file name, line number,
```



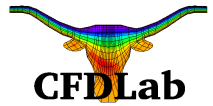
Poisson `main()` IV

```
    // and then abort.  Currrently the library does not use C++
    // exception handling.
    libmesh_error();
  }

// Brief message to the user regarding the program name
// and command line arguments.
else
  {
    std::cout << "Running " << argv[0];

    for (int i=1; i<argc; i++)
      std::cout << " " << argv[i];

    std::cout << std::endl << std::endl;
  }


// Read problem dimension from command line.  Use int
// instead of unsigned since the GetPot overload is ambiguous
// otherwise.
int dim = 2;
if ( command_line.search(1, "-d") )
  dim = command_line.next(dim);

// Skip higher-dimensional examples on a lower-dimensional libMesh build
libmesh_example_assert(dim <= LIBMESH_DIM, "2D/3D support");

// Create a mesh with user-defined dimension.
// Read number of elements from command line
```



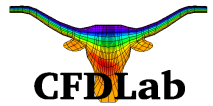
Poisson `main()` V

```
int ps = 15;
if ( command_line.search(1, "-n") )
  ps = command_line.next(ps);

// Read FE order from command line
std::string order = "SECOND";
if ( command_line.search(2, "-Order", "-o") )
  order = command_line.next(order);

// Read FE Family from command line
std::string family = "LAGRANGE";
if ( command_line.search(2, "-FEFamily", "-f") )
  family = command_line.next(family);

// Cannot use discontinuous basis.
if ((family == "MONOMIAL") || (family == "XYZ"))
  {
    if (libMesh::processor_id() == 0)
      std::cerr << "ex4 currently requires a C^0 (or higher) FE basis." << std::endl;
    libmesh_error();
  }

// Create a mesh, with dimension to be overridden later, distributed
// across the default MPI communicator.
Mesh mesh(init.comm());

// Use the MeshTools::Generation mesh generator to create a uniform
// grid on the square [-1,1]^D.  We instruct the mesh generator
// to build a mesh of 8x8 \p Quad9 elements in 2D, or \p Hex27
// elements in 3D.  Building these higher-order elements allows
```



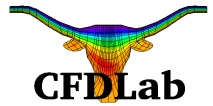
Poisson `main()` VI

```
// us to use higher-order approximation, as in example 3.

Real halfwidth = dim > 1 ? 1. : 0.;
Real halfheight = dim > 2 ? 1. : 0.;

if ((family == "LAGRANGE") && (order == "FIRST"))
  {
    // No reason to use high-order geometric elements if we are
    // solving with low-order finite elements.
    MeshTools::Generation::build_cube (mesh,
                                       ps,
                                       (dim>1) ? ps : 0,
                                       (dim>2) ? ps : 0,
                                       -1., 1.,
                                       -halfwidth, halfwidth,
                                       -halfheight, halfheight,
                                       (dim==1)    ? EDGE2 :
                                       ((dim == 2) ? QUAD4 : HEX8));
  }

else
  {
    MeshTools::Generation::build_cube (mesh,
                                       ps,
                                       (dim>1) ? ps : 0,
                                       (dim>2) ? ps : 0,
                                       -1., 1.,
                                       -halfwidth, halfwidth,
                                       -halfheight, halfheight,
                                       (dim==1)    ? EDGE3 :
```



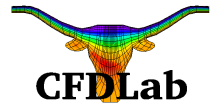
Poisson `main()` VII

```
                                          ((dim == 2) ? QUAD9 : HEX27));
  }


// Print information about the mesh to the screen.
mesh.print_info();


// Create an equation systems object.
EquationSystems equation_systems (mesh);

// Declare the system and its variables.
// Create a system named "Poisson"
LinearImplicitSystem& system =
  equation_systems.add_system<LinearImplicitSystem> ("Poisson");


// Add the variable "u" to "Poisson".  "u"
// will be approximated using second-order approximation.
unsigned int u_var = system.add_variable("u",
                                         Utility::string_to_enum<Order>   (order),
                                         Utility::string_to_enum<FEFamily>(family));

Poisson poisson(equation_systems);

// Give the system a pointer to the matrix assembly
// function.
system.attach_assemble_object (poisson);

// Construct a Dirichlet boundary condition object
```



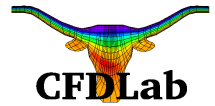
Poisson `main()` VIII

```
// Indicate which boundary IDs we impose the BC on
// We either build a line, a square or a cube, and
// here we indicate the boundaries IDs in each case
std::set<boundary_id_type> boundary_ids;
// the dim==1 mesh has two boundaries with IDs 0 and 1
boundary_ids.insert(0);
boundary_ids.insert(1);
// the dim==2 mesh has four boundaries with IDs 0, 1, 2 and 3
if(dim>=2)
{
  boundary_ids.insert(2);
  boundary_ids.insert(3);
}
// the dim==3 mesh has four boundaries with IDs 0, 1, 2, 3, 4 and 5
if(dim==3)
{
  boundary_ids.insert(4);
  boundary_ids.insert(5);
}

// Create a vector storing the variable numbers which the BC applies to
std::vector<unsigned int> variables(1);
variables[0] = u_var;

// Create an AnalyticFunction object that we use to project the BC
// This function just calls the function exact_solution via exact_solution_wrapper
AnalyticFunction<> exact_solution_object(exact_solution_wrapper);

DirichletBoundary dirichlet_bc(boundary_ids,
```



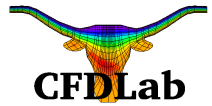
Poisson `main()` IX

```
                                variables,
                                &exact_solution_object);

  // We must add the Dirichlet boundary condition _before_
  // we call equation_systems.init()
  system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);


  // Initialize the data structures for the equation system.
  equation_systems.init();

  // Print information about the system to the screen.
  equation_systems.print_info();
  mesh.print_info();

  // Solve the system "Poisson", just like example 2.
  system.solve();

  // After solving the system write the solution
  // to a GMV-formatted plot file.
  if(dim == 1)
  {
    GnuPlotIO plot(mesh,"Introduction Example 4, 1D",GnuPlotIO::GRID_ON);
    plot.write_equation_systems("gnuplot_script",equation_systems);
  }
#ifdef LIBMESH_HAVE_EXODUS_API
  else
  {
    ExodusII_IO (mesh).write_equation_systems ((dim == 3) ?
      "out_3.e" : "out_2.e",equation_systems);
```



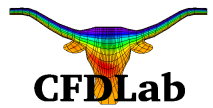
Poisson `main()` X

```
  }
#endif // #ifdef LIBMESH_HAVE_EXODUS_API

  // All done.
  return 0;
}
```



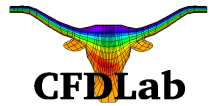
Poisson `assembly()` I

```
#include "poisson_problem.h"



void Poisson::assemble ()
{
  // Declare a performance log.  Give it a descriptive
  // string to identify what part of the code we are
  // logging, since there may be many PerfLogs in an
  // application.
  PerfLog perf_log ("Matrix Assembly");

    // Get a constant reference to the mesh object.
  const MeshBase& mesh = es.get_mesh();

  // The dimension that we are running
  const unsigned int dim = mesh.mesh_dimension();

  // Get a reference to the LinearImplicitSystem we are solving
  LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Poisson");

  // A reference to the \p DofMap object for this system.  The \p DofMap
  // object handles the index translation from node and element numbers
  // to degree of freedom numbers.  We will talk more about the \p DofMap
  // in future examples.
  const DofMap& dof_map = system.get_dof_map();

  // Get a constant reference to the Finite Element type
  // for the first (and only) variable in the system.
  FEType fe_type = dof_map.variable_type(0);
```



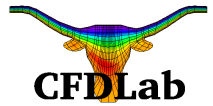
Poisson `assembly()` II

```
// Build a Finite Element object of the specified type.  Since the
// \p FEBase::build() member dynamically creates memory we will
// store the object as an \p AutoPtr<FEBase>.  This can be thought
// of as a pointer that will clean up after itself.
AutoPtr<FEBase> fe (FEBase::build(dim, fe_type));

// A 5th order Gauss quadrature rule for numerical integration.
QGauss qrule (dim, FIFTH);

// Tell the finite element object to use our quadrature rule.
fe->attach_quadrature_rule (&qrule);

// Declare a special finite element object for
// boundary integration.
AutoPtr<FEBase> fe_face (FEBase::build(dim, fe_type));

// Boundary integration requires one quadraure rule,
// with dimensionality one less than the dimensionality
// of the element.
QGauss qface(dim-1, FIFTH);

// Tell the finte element object to use our
// quadrature rule.
fe_face->attach_quadrature_rule (&qface);

// Here we define some references to cell-specific data that
// will be used to assemble the linear system.
// We begin with the element Jacobian * quadrature weight at each
// integration point.
```



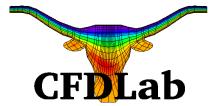
Poisson `assembly()` III

```
const std::vector<Real>& JxW = fe->get_JxW();

// The physical XY locations of the quadrature points on the element.
// These might be useful for evaluating spatially varying material
// properties at the quadrature points.
const std::vector<Point>& q_point = fe->get_xyz();

// The element shape functions evaluated at the quadrature points.
const std::vector<std::vector<Real> >& phi = fe->get_phi();

// The element shape function gradients evaluated at the quadrature
// points.
const std::vector<std::vector<RealGradient> >& dphi = fe->get_dphi();

// Define data structures to contain the element matrix
// and right-hand-side vector contribution.  Following
// basic finite element terminology we will denote these
// "Ke" and "Fe". More detail is in example 3.
DenseMatrix<Number> Ke;
DenseVector<Number> Fe;

// This vector will hold the degree of freedom indices for
// the element.  These define where in the global system
// the element degrees of freedom get mapped.
std::vector<dof_id_type> dof_indices;

// Now we will loop over all the elements in the mesh.
// We will compute the element matrix and right-hand-side
// contribution.  See example 3 for a discussion of the
// element iterators.
```



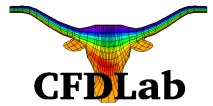
Poisson `assembly()` IV

```
MeshBase::const_element_iterator       el     = mesh.active_local_elements_begin();
const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();

for ( ; el != end_el; ++el)
  {
    // Start logging the shape function initialization.
    // This is done through a simple function call with
    // the name of the event to log.
    perf_log.push("elem init");

    // Store a pointer to the element we are currently
    // working on.  This allows for nicer syntax later.
    const Elem* elem = *el;

    // Get the degree of freedom indices for the
    // current element.  These define where in the global
    // matrix and right-hand-side this element will
    // contribute to.
    dof_map.dof_indices (elem, dof_indices);

    // Compute the element-specific data for the current
    // element.  This involves computing the location of the
    // quadrature points (q_point) and the shape functions
    // (phi, dphi) for the current element.
    fe->reinit (elem);

    // Zero the element matrix and right-hand side before
    // summing them.  We use the resize member here because
    // the number of degrees of freedom might have changed from
    // the last element.  Note that this will be the case if the
```



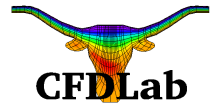
Poisson `assembly()` V

```
// element type is different (i.e. the last element was a
// triangle, now we are on a quadrilateral).
Ke.resize (dof_indices.size(),
           dof_indices.size());

Fe.resize (dof_indices.size());

// Stop logging the shape function initialization.
// If you forget to stop logging an event the PerfLog
// object will probably catch the error and abort.
perf_log.pop("elem init");

// Now we will build the element matrix.  This involves
// a double loop to integrate the test funcions (i) against
// the trial functions (j).
//
// We have split the numeric integration into two loops
// so that we can log the matrix and right-hand-side
// computation seperately.
//
// Now start logging the element matrix computation
perf_log.push ("Ke");

for (unsigned int qp=0; qp<qrule.n_points(); qp++)
  for (unsigned int i=0; i<phi.size(); i++)
    for (unsigned int j=0; j<phi.size(); j++)
      Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);


// Stop logging the matrix computation
```



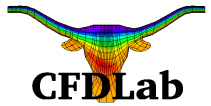
Poisson `assembly()` VI

```
      perf_log.pop ("Ke");

      // Now we build the element right-hand-side contribution.
      // This involves a single loop in which we integrate the
      // "forcing function" in the PDE against the test functions.
      //
      // Start logging the right-hand-side computation
      perf_log.push ("Fe");

      for (unsigned int qp=0; qp<qrule.n_points(); qp++)
        {
          // fxy is the forcing function for the Poisson equation.
          // In this case we set fxy to be a finite difference
          // Laplacian approximation to the (known) exact solution.
          //
          // We will use the second-order accurate FD Laplacian
          // approximation, which in 2D on a structured grid is
          //
          // u_xx + u_yy = (u(i-1,j) + u(i+1,j) +
          //                u(i,j-1) + u(i,j+1) +
          //                -4*u(i,j))/h^2
          //
          // Since the value of the forcing function depends only
          // on the location of the quadrature point (q_point[qp])
          // we will compute it here, outside of the i-loop
          const Real x = q_point[qp](0);
#if LIBMESH_DIM > 1
          const Real y = q_point[qp](1);
#else
          const Real y = 0.;
```



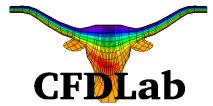
Poisson `assembly()` VII

```
#endif
#if LIBMESH_DIM > 2
          const Real z = q_point[qp](2);
#else
          const Real z = 0.;
#endif
          const Real eps = 1.e-3;

          const Real uxx = (exact_solution(x-eps,y,z) +
                            exact_solution(x+eps,y,z) +
                            -2.*exact_solution(x,y,z))/eps/eps;

          const Real uyy = (exact_solution(x,y-eps,z) +
                            exact_solution(x,y+eps,z) +
                            -2.*exact_solution(x,y,z))/eps/eps;

          const Real uzz = (exact_solution(x,y,z-eps) +
                            exact_solution(x,y,z+eps) +
                            -2.*exact_solution(x,y,z))/eps/eps;

          Real fxy;
          if(dim==1)
          {
            // In 1D, compute the rhs by differentiating the
            // exact solution twice.
            const Real pi = libMesh::pi;
            fxy = (0.25*pi*pi)*sin(.5*pi*x);
          }
          else
          {
```



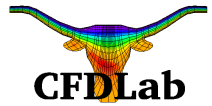
Poisson `assembly()` VIII

```
        fxy = - (uxx + uyy + ((dim==2) ? 0. : uzz));
      }

      // Add the RHS contribution
      for (unsigned int i=0; i<phi.size(); i++)
        Fe(i) += JxW[qp]*fxy*phi[i][qp];
    }

  // Stop logging the right-hand-side computation
  perf_log.pop ("Fe");

  // If this assembly program were to be used on an adaptive mesh,
  // we would have to apply any hanging node constraint equations
  // Also, note that here we call heterogenously_constrain_element_matrix_and_vector
  // to impose a inhomogeneous Dirichlet boundary conditions.
  dof_map.heterogenously_constrain_element_matrix_and_vector (Ke, Fe, dof_indices);

  // The element matrix and right-hand-side are now built
  // for this element.  Add them to the global matrix and
  // right-hand-side vector.  The \p SparseMatrix::add_matrix()
  // and \p NumericVector::add_vector() members do this for us.
  // Start logging the insertion of the local (element)
  // matrix and vector into the global matrix and vector
  perf_log.push ("matrix insertion");

  system.matrix->add_matrix (Ke, dof_indices);
  system.rhs->add_vector    (Fe, dof_indices);

  // Start logging the insertion of the local (element)
  // matrix and vector into the global matrix and vector
```



Poisson `assembly()` IX

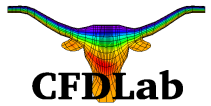
```
      perf_log.pop ("matrix insertion");
    }

  // That's it.  We don't need to do anything else to the
  // PerfLog.  When it goes out of scope (at this function return)
  // it will print its log to the screen. Pretty easy, huh?
}
```



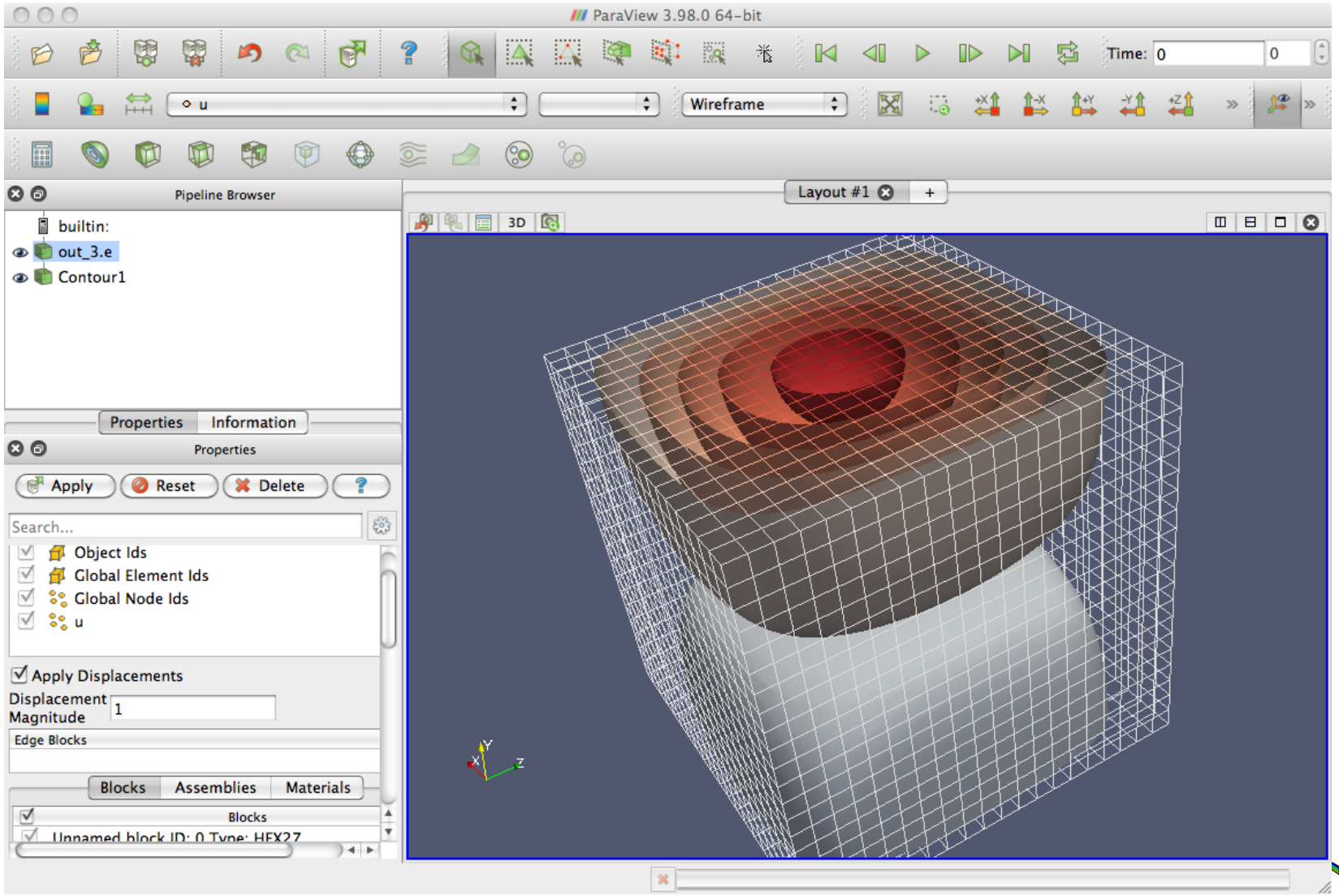
Running the program

Running the program

```
# copy & build the example
$ cp -r $LIBMESH_TUTORIAL/poisson .
$ cd poisson
$ make

# run the example in 2D with 20 elements in each direction
$ ./example-opt -d 2 -n 20

# run the example in 3D with 20 elements in each direction
$ ./example-opt -d 3 -n 20

# '' '' except using the Trilinos linear solvers
$ ./example-opt -d 3 -n 20 --use-trilinos
```



Output

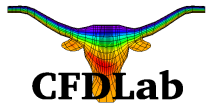

ParaView 3.98.0 64-bit
Time: 0
0
u
Wireframe
Pipeline Browser
builtin:
out_3.e
Contour1
Properties
Information
Properties
Apply
Reset
Delete
Search...
Object Ids
Global Element Ids
Global Node Ids
u
Apply Displacements
Displacement Magnitude
1
Edge Blocks
Blocks
Assemblies
Materials
Blocks
Layout #1
3D

Extension: Multithreaded Assembly:

`poisson_threaded`

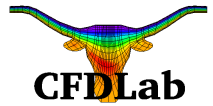
Multithreading in `libMesh`

- The `libMesh::Threads::()` namespace provides a transparent wrapper for Intel's Threading Building Blocks.
- This is used extensively inside the library to speed up compute-heavy operations.
  - to use it, make sure you are compiled with `--enable-tbb` and add `--n_threads=#` to the command line.
- To get the most benefit from threads you'll want to write a threaded assembly routine as well.
- Fortunately, this is easy.



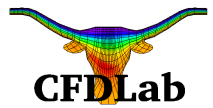
Poisson class definition

```
// headers omitted for brevity
class Poisson : public System::Assembly
{
public:
  Poisson (EquationSystems &es_in) :
    es (es_in)
  {}

  void assemble ();

  void operator()(const ConstElemRange &range) const;

  Real exact_solution (const Real x,
                       const Real y,
                       const Real z = 0.) const
  {
    static const Real pi = acos(-1.);

    return cos(.5*pi*x)*sin(.5*pi*y)*cos(.5*pi*z);
  }

private:
  EquationSystems &es;

  mutable Threads::spin_mutex assembly_mutex;
};
```



Threaded Poisson assembly

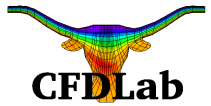
```
#include "poisson_problem.h"

void Poisson::assemble ()
{
  const MeshBase& mesh = es.get_mesh();

  ConstElemRange assembly_elem_range (mesh.active_local_elements_begin(),
                                      mesh.active_local_elements_end());

  Threads::parallel_for (// the range over which we will perform threaded operations
                         assembly_elem_range,

                         // the function object to apply to each element in the range
                         *this);
}

void Poisson::operator()(const ConstElemRange &range) const
{
  ...

  // insert the local (per-thread) element matrix/vector into
  // the global matrix/vector.  This is a shared object, so we
  // must be careful to lock for exclusive access.
  {
    Threads::spin_mutex::scoped_lock lock(assembly_mutex);

    system.matrix->add_matrix (Ke, dof_indices);
    system.rhs->add_vector    (Fe, dof_indices);
  }
}
```

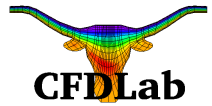


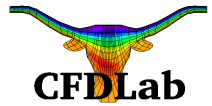
Running the program

Running the program

```
# copy & build the example
$ cp -r $LIBMESH_TUTORIAL/poisson_threaded .
$ cd poisson_threaded
$ make

# run the example in 2D with 20 elements in each direction
$ ./example-opt -d 2 -n 20

# run the example in 3D with 20 elements in each direction
# using various numbers of threads
$ ./example-opt -d 3 -n 20 --n_threads=1
$ ./example-opt -d 3 -n 20 --n_threads=2
$ ./example-opt -d 3 -n 20 --n_threads=4
```



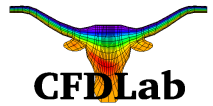




- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \boldsymbol{b} \cdot \nabla u = f$$

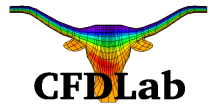
```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        +(b*dphi[j][q])*phi[i][q]);
  }
```



- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \boldsymbol{b} \cdot \nabla u = f$$

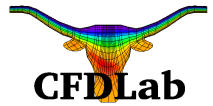
```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        +(b*dphi[j][q])*phi[i][q]);
  }
```



- The matrix assembly routine for the linear convection-diffusion equation,

$$-k\Delta u + \boldsymbol{b} \cdot \nabla u = f$$

```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)   += JxW[q]*f(xyz[q])*phi[i][q];

    for (j=0; j<Ns; ++j)
      Ke(i,j) += JxW[q]*(k*(dphi[j][q]*dphi[i][q])
                        +(b*dphi[j][q])*phi[i][q]);
  }
```



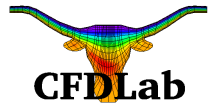
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- We have an array of submatrices: `Ke[ ][ ]`



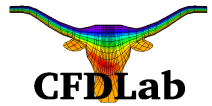
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- We have an array of submatrices: `Ke[0][0]`



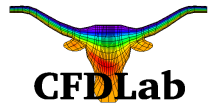
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- We have an array of submatrices: `Ke[1][1]`



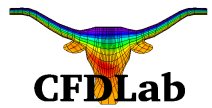
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- We have an array of submatrices: `Ke[2][1]`



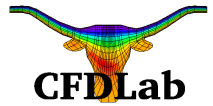
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- And an array of right-hand sides: `Fe[]`.



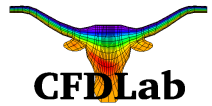
- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \qquad \in \quad \Omega \subset \mathbb{R}^2$$

- The element stiffness matrix concept can extended to include sub-matrices

$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

- And an array of right-hand sides: `Fe[0]`.



- For multi-variable systems like Stokes flow,

$$\begin{aligned} \nabla p - \nu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0 \end{aligned} \quad \in \quad \Omega \subset \mathbb{R}^2$$

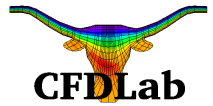
- The element stiffness matrix concept can extended to include sub-matrices

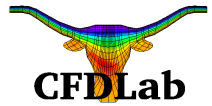
$$\left[\begin{array}{cc|c} K^e_{u_1u_1} & K^e_{u_1u_2} & K^e_{u_1p} \\ K^e_{u_2u_1} & K^e_{u_2u_2} & K^e_{u_2p} \\ \hline K^e_{pu_1} & K^e_{pu_2} & K^e_{pp} \end{array}\right] \left[\begin{array}{c} U^e_{u_1} \\ U^e_{u_2} \\ \hline U^e_p \end{array}\right] - \left[\begin{array}{c} F^e_{u_1} \\ F^e_{u_2} \\ \hline F^e_p \end{array}\right]$$

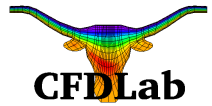
- And an array of right-hand sides: `Fe[1]`.

- The matrix assembly can proceed in essentially the same way.
- For the momentum equations:

```
for (q=0; q<Nq; ++q)
  for (d=0; d<2; ++d)
    for (i=0; i<Ns; ++i) {
      Fe[d](i) += JxW[q]*f(xyz[q],d)*phi[i][q];

      for (j=0; j<Ns; ++j)
        Ke[d][d](i,j) +=
            JxW[q]*nu*(dphi[j][q]*dphi[i][q]);
    }
```

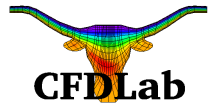






- For linear problems, we have already seen how the weighted residual statement leads directly to a sparse linear system of equations

$$\boldsymbol{KU} = \boldsymbol{F}$$



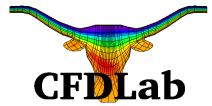
- For time-dependent problems,

$$\frac{\partial u}{\partial t} = F(u)$$

- we also need a way to advance the solution in time, e.g. a θ-method

$$\begin{aligned} \left(\frac{u^{n+1} - u^n}{\Delta t}, v^h \right) &= \left(F(u_\theta), v^h \right) \quad \forall v^h \in \mathcal{V}^h \\ u_\theta &:= \theta u^{n+1} + (1-\theta) u^n \end{aligned}$$

- Leads to $\boldsymbol{KU} = \boldsymbol{F}$ at *each timestep*.

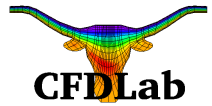


- For nonlinear problems, typically a sequence of linear problems must be solved, e.g. for Newton's method

$$(F'(u^k)\delta u^{k+1}, v) = -(F(u^k), v)$$

  where $F'(u^k)$ is the linearized (Jacobian) operator associated with the PDE.

- Must solve $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}\delta \boldsymbol{U} = -\boldsymbol{F}$ (Inexact Newton method) at *each iteration step*.



Examples: Nonlinear & Transient Problems

`laplace_young`

`transient_convection_diffusion`

`navier_stokes`

Lapace-Young "minimal surface" problem

The Laplace-Young equation governs the behavior of films, which seek to form a minimal surface:

$$-\nabla \cdot \left(\frac{\nabla u}{\sqrt{1 + \nabla u \cdot \nabla u}} \right) + \kappa u = 0$$

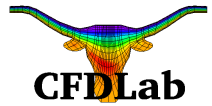
or equivalently

$$-\nabla \cdot \left(K\left(u\right) \nabla u \right) + \kappa u = 0$$

This problem behaves like a Helmholtz problem with nonlinear diffusion coefficient, $K(u)$.

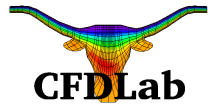
Laplace-Young Assembly

```
// headers omitted for brevity
class LaplaceYoung : public NonlinearImplicitSystem::ComputeJacobian,
                     public NonlinearImplicitSystem::ComputeResidual
{
public:
  LaplaceYoung (EquationSystems &es_in) :
    es(es_in)
  {}

  virtual void jacobian (const NumericVector<Number> &soln,
                         SparseMatrix<Number> &jacobian,
                         NonlinearImplicitSystem &system);

  virtual void residual (const NumericVector<Number> &soln,
                         NumericVector<Number> &resid,
                         NonlinearImplicitSystem &system);

private:
  EquationSystems &es;
};
```



Laplace-Young Assembly

```
// Get the degree of freedom indices for the
// current element.
dof_map.dof_indices (elem, dof_indices);

// Now we will build the element Jacobian.  This involves
// a double loop to integrate the test funcions (i) against
// the trial functions (j). Note that the Jacobian depends
// on the current solution x, which we access using the soln
// vector.
//
for (unsigned int qp=0; qp<qrule.n_points(); qp++)
  {
    Gradient grad_u;

    for (unsigned int i=0; i<phi.size(); i++)
      grad_u += dphi[i][qp]*soln(dof_indices[i]);

    const Number K = 1./std::sqrt(1. + grad_u*grad_u);
    ...
  }
```



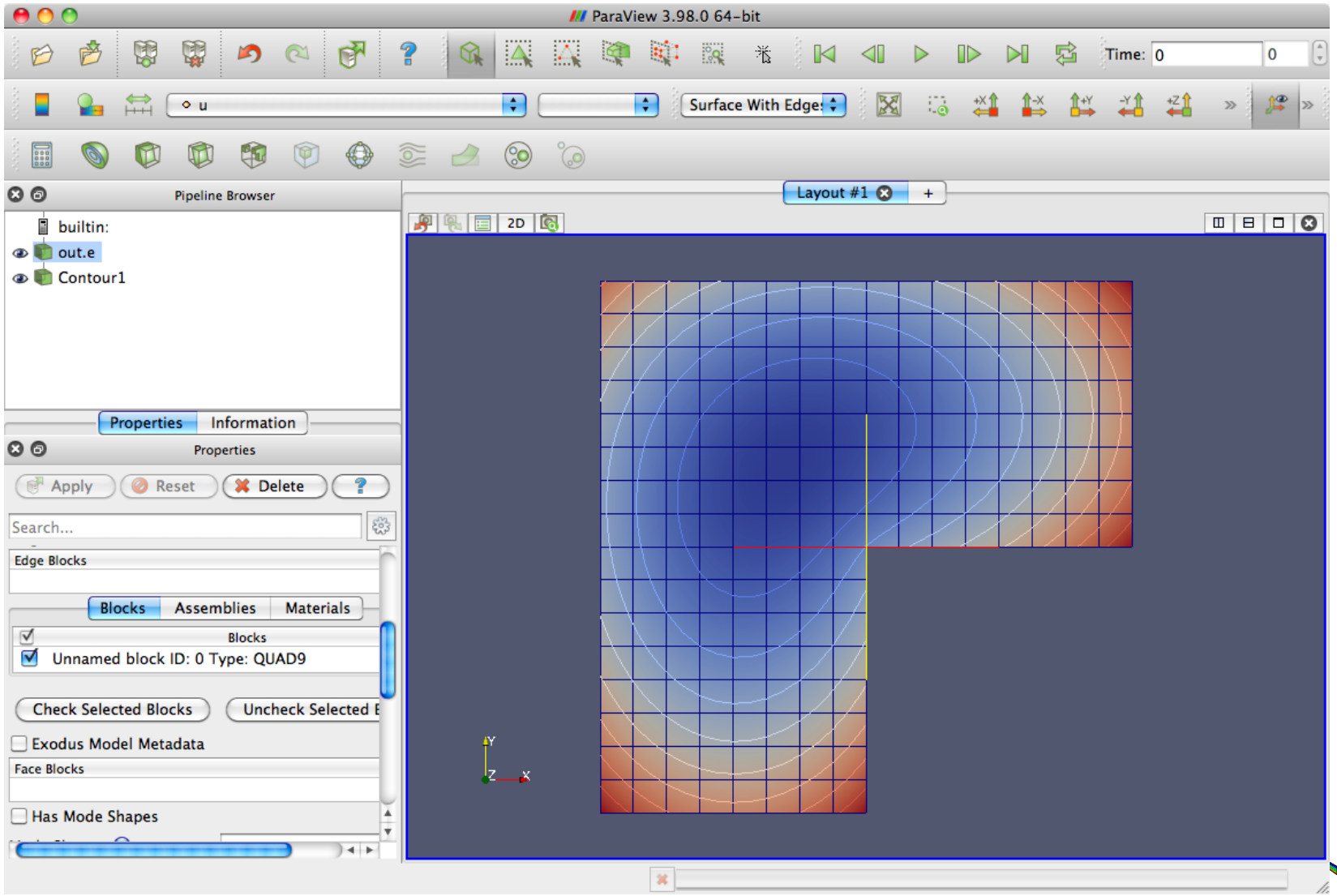
Running the `laplace_young` program

Running the program

```
# copy the example
$ cp -r $LIBMESH_TUTORIAL/laplace_young .
$ cd laplace_young
$ make

# run the example with 3 uniform refinement steps, using first
# order Lagrange elements
$ ./example-opt -r 3 -o FIRST

# run the example with 3 uniform refinement steps, using first
# order Lagrange elements
$ ./example-opt -r 3 -o SECOND
```

CFDLab

Output

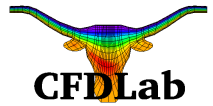


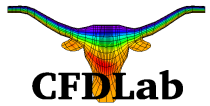
CFDLab

Running the `transient_convection_diffusion` program

Running the program

```
# copy the example
$ cp -r $LIBMESH_TUTORIAL/transient_convection_diffusion .
$ cd transient_convection_diffusion
$ make

# run the example
$ ./example-opt
```

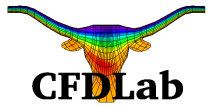



CFDLab

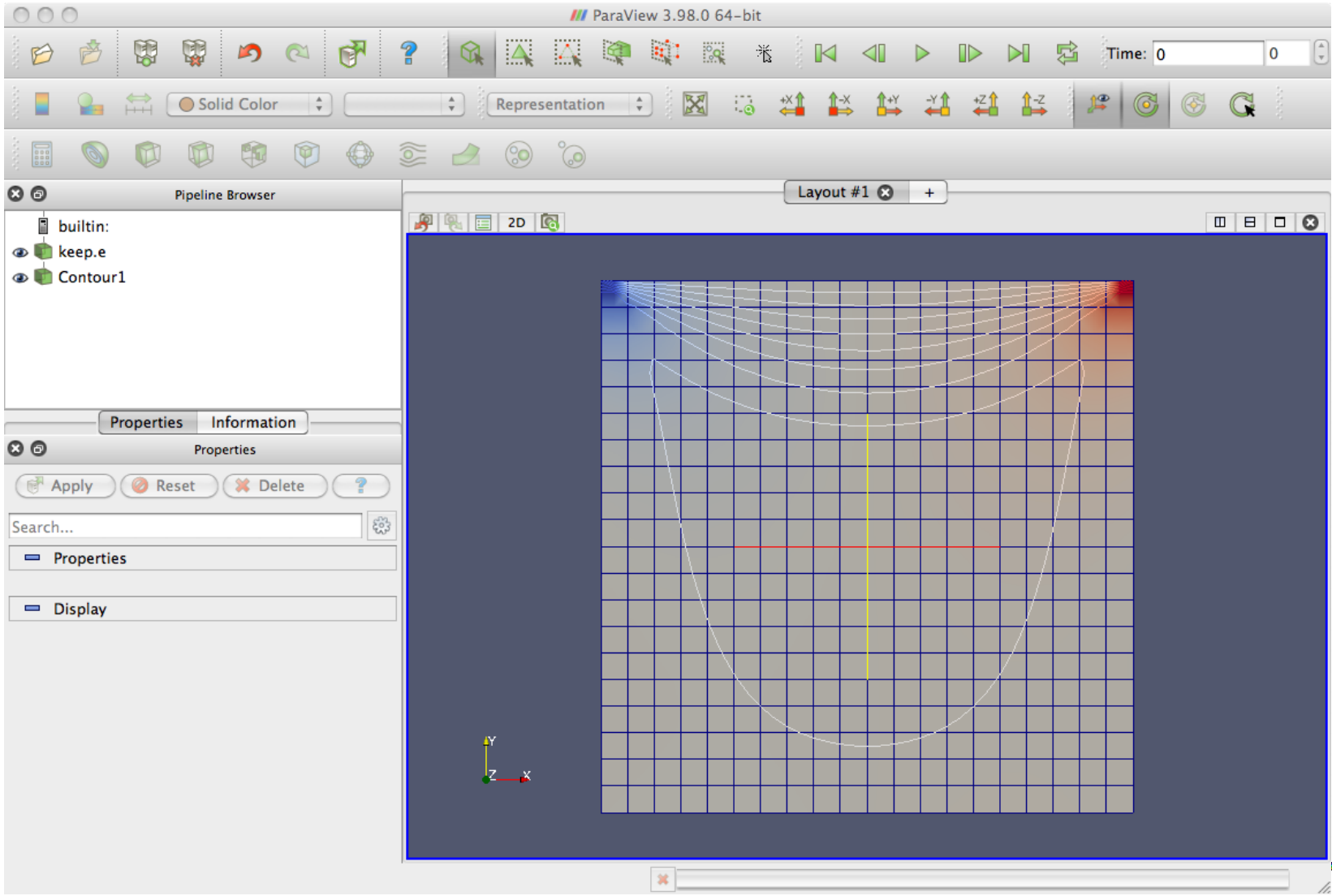
Running the navier_stokes program

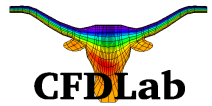
Running the program

```
# copy the example
$ cp -r $LIBMESH_TUTORIAL/navier_stokes .
$ cd navier_stokes
$ make

# run the example
$ ./example-opt
$ mpirun -np 2 ./example-opt
```

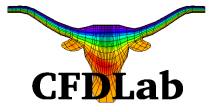


Output


ParaView 3.98.0 64-bit
Time: 0
0
Solid Color
Representation
Pipeline Browser
builtin:
keep.e
Contour1
Layout #1
2D
Properties
Information
Properties
Apply
Reset
Delete
Search...
Properties
Display
Y
Z
X



Adaptive Mesh Refinement



Model Problem

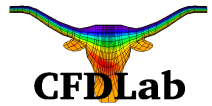
- Consider the 1D model ODE

$$\begin{cases} -u'' + bu' + cu = f \in \Omega = (0, L) \\ u(0) = u_0 \\ u(L) = u_L \end{cases} \tag{1}$$

- with weak form

$$\int_\Omega \left(u'v' + bu'v + cuv \right) \, dx = \int_\Omega fv \, dx \tag{2}$$

for every $v \in H_0^1(\Omega)$.



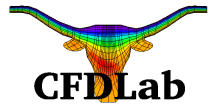
Model Problem (cont.)

- The analogous d-dimensional problem with $\Omega \subset \mathbb{R}^d$ and boundary $\partial\Omega$ is

$$\begin{cases} -\Delta u + \boldsymbol{b} \cdot \nabla u + cu &= f &\in \Omega \\ u &= g &\in \partial\Omega \end{cases} \tag{3}$$

- with weak form

$$\int_\Omega (\nabla u \cdot \nabla v + (\boldsymbol{b} \cdot \nabla u)v + cuv)\ dx = \int_\Omega fv\ dx \tag{4}$$

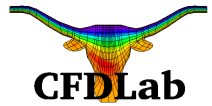


Model Problem (cont.)

- The finite element method works with the weak form, replacing the trial and test functions u, v with their approximations u^h, v^h, and summing the contributions of the element integrals

$$\sum_{e=1}^{N_e} \int_{\Omega_e} \left(\nabla u^h \cdot \nabla v^h + (\boldsymbol{b} \cdot \nabla u^h) v^h + c u^h v^h - f v^h \right) \, dx = 0 \qquad (5)$$

- Remark: We considered here a standard piecewise continuous finite element basis. In general, ∇u^h will have a jump discontinuity across element boundaries.



Galerkin FE Method

- Expressing u^h and v^h in our chosen piecewise continuous polynomial basis

$$u^h = \sum_{j=1}^{N} u_j \varphi_j \qquad v^h = \sum_{i=1}^{N} c_i \varphi_i \tag{6}$$

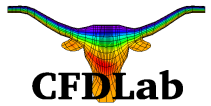
we obtain on each element Ω_e

$$\sum_{j=1}^{N} u_j \left[\int_{\Omega_e} (\nabla \varphi_j \cdot \nabla \varphi_i + (\boldsymbol{b} \cdot \nabla \varphi_j)\varphi_i + c\varphi_j \varphi_i)\, dx \right] = \int_{\Omega_e} f \varphi_i \, dx \tag{7}$$

for $i = 1 \ldots N$.

- In the standard element-stiffness matrix form,

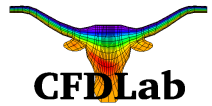
$$\boldsymbol{K_e U} = \boldsymbol{F_e} \tag{8}$$



LibMesh Representation

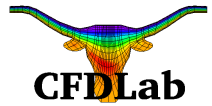
- To code the model problem in `LibMesh`, the user must provide the routine which computes $\boldsymbol{K_e}$ and $\boldsymbol{F_e}$ for each element.
- The integrals are computed over the reference element using an appropriate numerical quadrature rule with weights w_q and points $\boldsymbol{\xi}_q$, $q = 1 \ldots N_q$.

$$\int_{\Omega_e} f\varphi_i \, dx = \int_{\hat{\Omega}_e} f\varphi_i \, |J| d\xi \approx \sum_{q=1}^{N_q} w_q |J(\boldsymbol{\xi}_q)| f(\boldsymbol{\xi}_q) \varphi_i(\boldsymbol{\xi}_q) \tag{9}$$



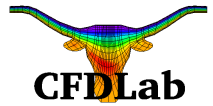
LibMesh Representation (cont.)

- `LibMesh` provides the following variables for constructing $\boldsymbol{K_e}$ and $\boldsymbol{F_e}$ at quadrature point `q`:
  - `JxW[q]` = the scalar value of the element Jacobian map times the quadrature rule weight
  - `phi[i][q]` = $\varphi_i(\boldsymbol{\xi}_q)$
  - `dphi[i][q]` = $(J^{-1} \cdot \nabla_{\boldsymbol{\xi}} \varphi_i)(\boldsymbol{\xi}_q)$ (e.g. in 1D, this is $\frac{\partial \phi_i}{\partial \xi} \frac{\partial \xi}{\partial x}(\xi_q)$)



LibMesh Representation (cont.)

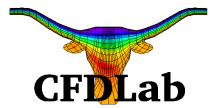
```
for (q=0; q<Nq; ++q) {
  // Compute b, c, f at this quadrature point
  // ...

  for (i=0; i<N; ++i) {
    Fe(i)   += JxW[q]*f*phi[i][q];

    for (j=0; j<N; ++j)
      Ke(i,j) += JxW[q]*(
        (dphi[i][q]*dphi[j][q])  +
        (b*dphi[j][q])*phi[i][q] +
         c*phi[j][q]*phi[i][q]
                          );
  }
}
```



Boundary Conditions

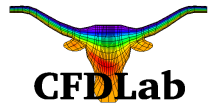
- Dirichlet boundary conditions are typically enforced after the global stiffness matrix $\mathbf{K}$ has been assembled
- This usually involves
  1. placing a "1" on the main diagonal of the global stiffness matrix
  2. zeroing out the row entries
  3. placing the Dirichlet value in the rhs vector
  4. subtracting off the column entries from the rhs

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & . \\ k_{21} & k_{22} & k_{23} & . \\ k_{31} & k_{32} & k_{33} & . \\ . & . & . & . \end{bmatrix}, \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ . \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & k_{22} & k_{23} & . \\ 0 & k_{32} & k_{33} & . \\ 0 & . & . & . \end{bmatrix}, \begin{bmatrix} g_1 \\ f_2 - k_{21} g_1 \\ f_3 - k_{31} g_1 \\ . \end{bmatrix}$$



Boundary Conditions (cont.)

- This approach works for an interpolary finite element basis but not in the general case.
- For large problems with parallel sparse matrices, it is inefficient to change individual entries once the global matrix is assembled.
- What is required is a way to enforce boundary conditions for a generic finite element basis *at the element stiffness matrix level*.
- A Solution: “Penalty” Boundary Conditions
  - *For many years, penalty boundary conditions were the preferred approach in* `libMesh`*, so you may encounter them.*

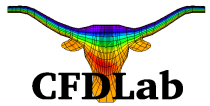


Penalty Boundary Conditions

- An additional "penalty" term is added to the standard weak form

$$\int_\Omega (\nabla u \cdot \nabla v + (\boldsymbol{b} \cdot \nabla u)v + cuv)\, dx + \underbrace{\frac{1}{\epsilon}\int_{\partial\Omega} (u-g)v\, dx}_{\text{penalty term}} = \int_\Omega fv\, dx$$

- Here $\epsilon \ll 1$ is chosen so that, in floating point arithmetic, $\frac{1}{\epsilon} + 1 = \frac{1}{\epsilon}$.
- This weakly enforces $u = g$ on the boundary at the element level, and works for general finite element bases. This approach only impacts elements who have a face on the boundary of the domain.

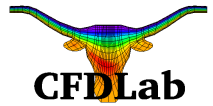


LibMesh Representation

`LibMesh` provides:

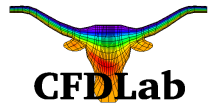
- A quadrature rule with `Nqf` points and `JxW_f[]`
- A finite element coincident with the boundary face that has `Nf` shape function values `phi_f[][]`

```
for (qf=0; qf<Nqf; ++qf) {
  // Compute g at this face quadrature point

  for (i=0; i<Nf; i++) {
    Fe(i) += JxW_f[qf]*penalty*g*phi_face[i][qf];

    for (j=0; j<Nf; j++)
      Ke(i,j) += JxW_f[qf]*penalty*phi_f[i][qf]*phi_f[j][qf];
  }
}
```

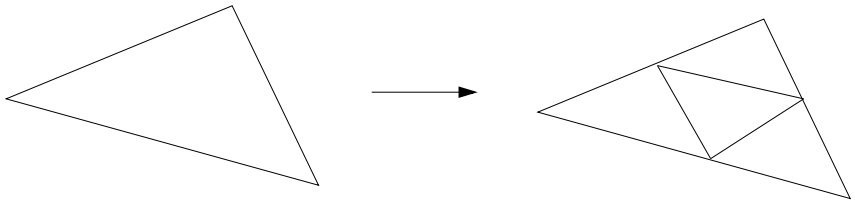


Adaptivity And Error Indicators

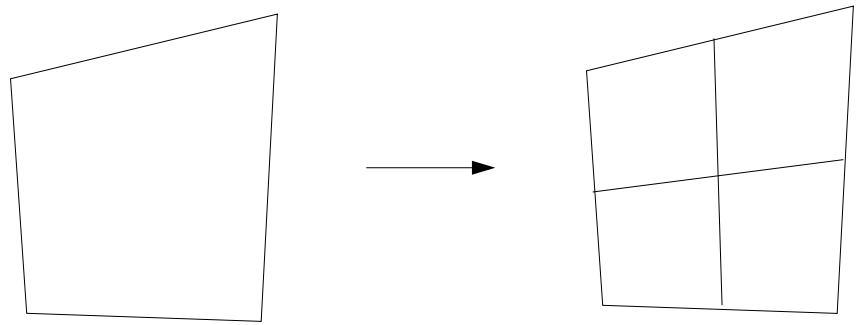
- A major goal of the `LibMesh` library is to provide:
  - Adaptive mesh refinement support for standard geometric elements
  - Generic, physics-independent error indicators
- In this context, we'll discuss
  - "Natural" refinement patterns
  - A flux-jump error indicator



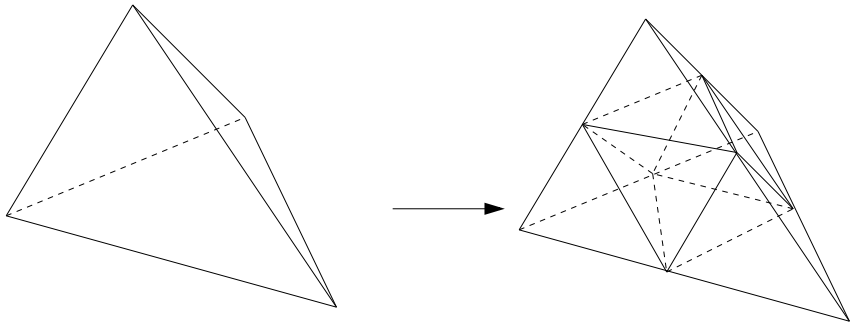
Natural Refinement Patterns



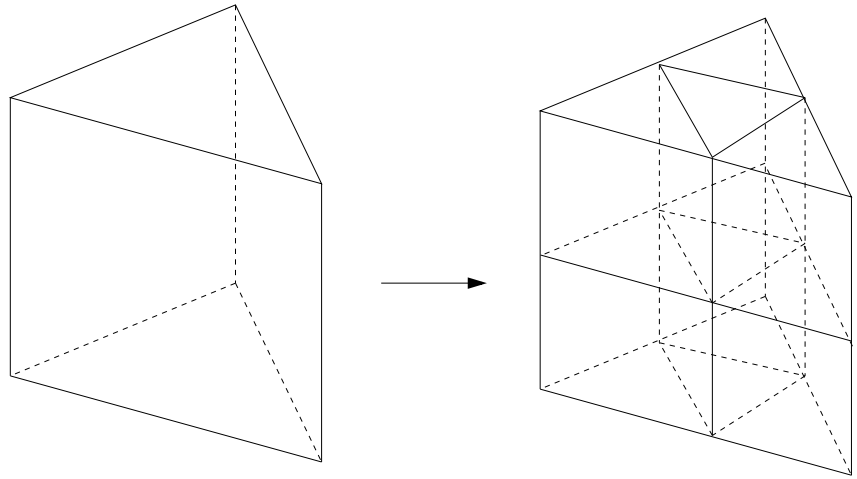
Triangle



Quadrilateral



Tetrahedron



Prism

Flux-Jump Error Indicator

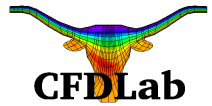
- The flux-jump error indicator is derived starting from the element integrals

$$\sum_{e=1}^{N_e} \int_{\Omega_e} \left(\nabla u^h \cdot \nabla v^h + (\boldsymbol{b} \cdot \nabla u^h) v^h + c u^h v^h - f v^h \right) \, dx = 0 \tag{5}$$

- Applying the divergence theorem "in reverse" obtains

$$\sum_{e=1}^{N_e} \int_{\Omega_e} \left(-\Delta u^h + (\boldsymbol{b} \cdot \nabla u^h) + c u^h - f \right) v^h \, dx + \tag{10}$$

$$\sum_{\partial\Omega_e \not\subset \partial\Omega} \int_{\partial\Omega_e} \left[\!\left[\frac{\partial u^h}{\partial n} \right]\!\right] v^h \, dx = 0$$



Flux-Jump Error Indicator (cont.)

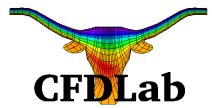
- Defining the cell residual

$$r(u^h) = -\Delta u^h + (\boldsymbol{b} \cdot \nabla u^h) + c u^h - f \tag{11}$$

we have

$$\sum_{e=1}^{N_e} \int_{\Omega_e} r(u^h) v^h \, dx + \sum_{\partial\Omega_e \not\subset \partial\Omega} \int_{\partial\Omega_e} \left[\!\left[\frac{\partial u^h}{\partial n} \right]\!\right] v^h \, dx = 0 \tag{12}$$

- Clearly, the exact solution u satisfies (12) identically.
- Computing $r(u^h)$ requires knowledge of the differential operator (i.e. knowledge of the "physics").
- The second sum leads to a *physics-independent* method for estimating the error in the approximate solution u^h.



Flux-Jump Error Indicator (cont.)

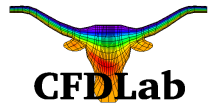
- Pros
  - Ideal for low-order (piecewise linear) elements
  - Easily extensible to adaptivity with hanging nodes
  - Works well in practice for nonlinear, time-dependent problems, and problems with shocks, layers, discontinuities, etc.
- Cons
  - For higher-order elements, the interior residual term may dominate
  - Relatively expensive to compute
  - Makes no sense for discontinuous and C^1 FE bases

1D Example

- In 1 dimension, the jump integrals reduce to point-wise evaluation of the derivatives at the element boundaries.
- For linear elements, the error indicator η for a particular element $\Omega_e = (x_e, x_{e+1})$ is defined as

$$\eta^2 = \frac{h_e}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} [\![u'(y_i)]\!]^2 \tag{13}$$

where $h_e = x_{e+1} - x_e$ is the element length, and $N_{\text{int}} \leq 2$ is the number of *interior* nodes y_i the element has.

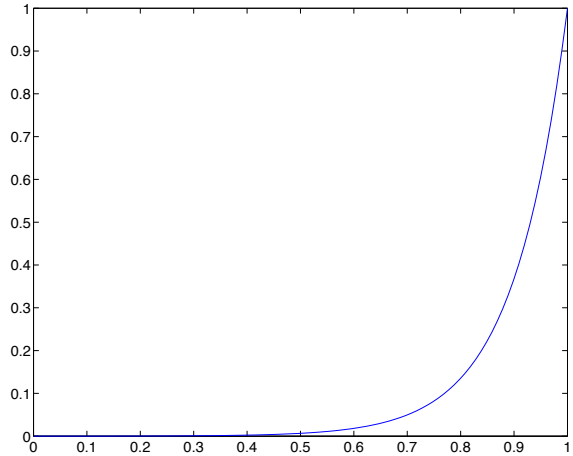


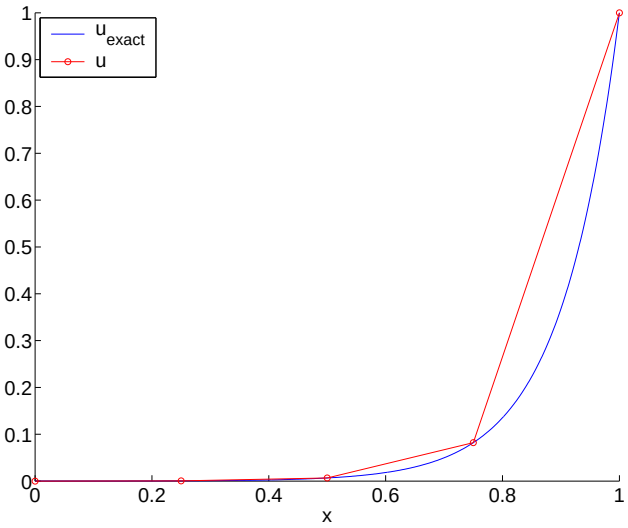
1D Example (cont.)

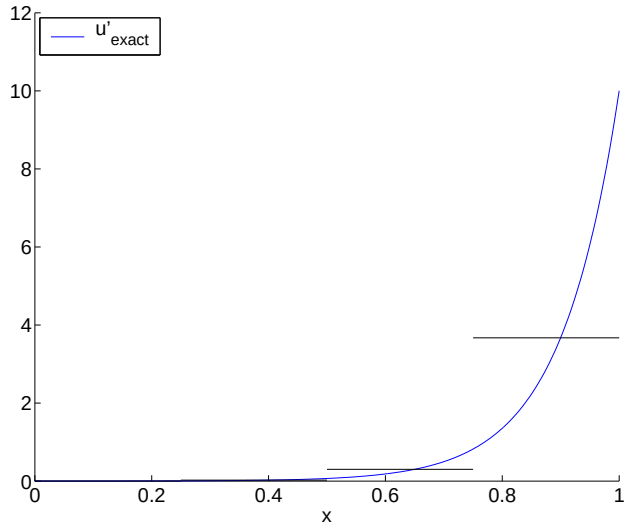
- Consider the function

$$u = \frac{1 - \exp(10x)}{1 - \exp(10)}$$

  which is a solution of the classic 1D advection-diffusion boundary layer equation.
- We assume here that the finite element solution is the linear interpolant of u, and compute the error indicator for a sequence of uniformly refined grids.

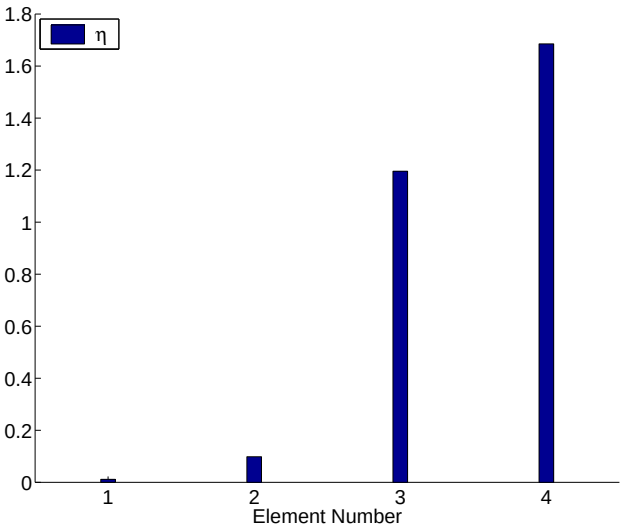


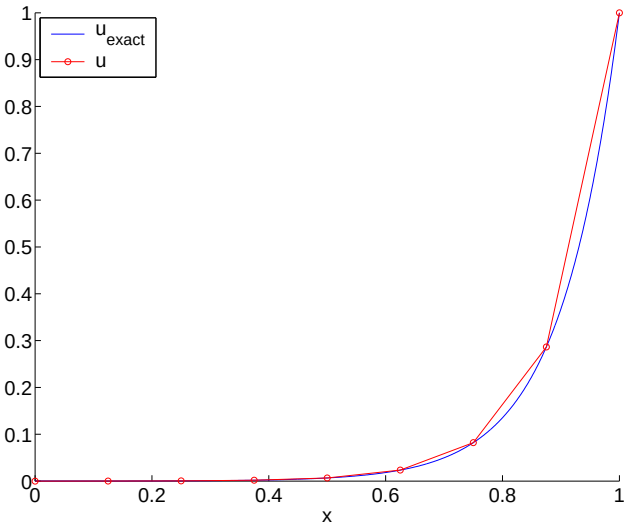


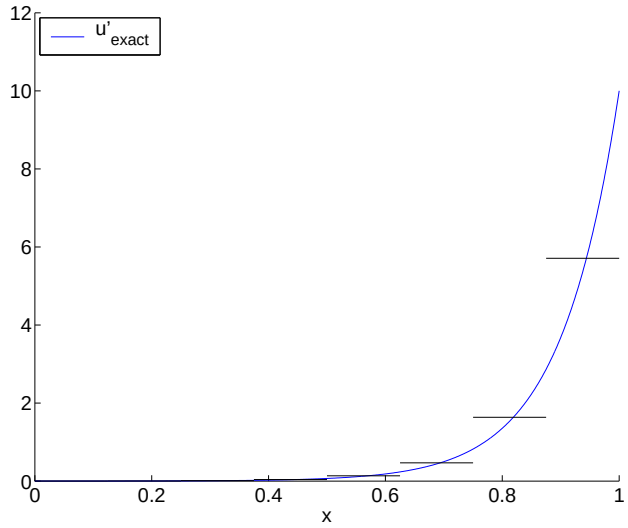


4 elements

$$||e||_{L_2} = 0.09$$

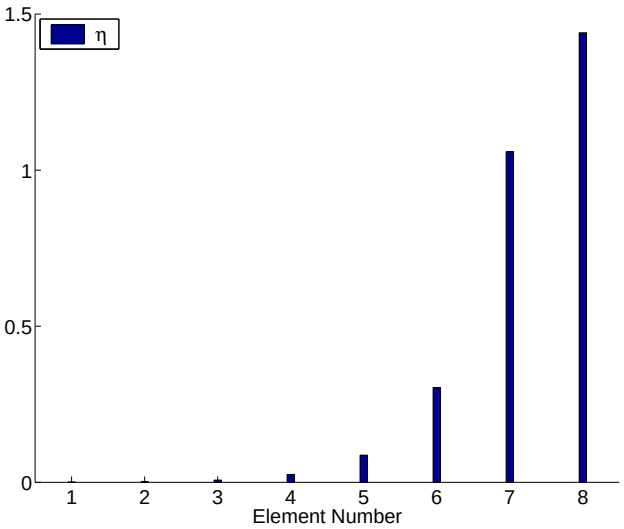


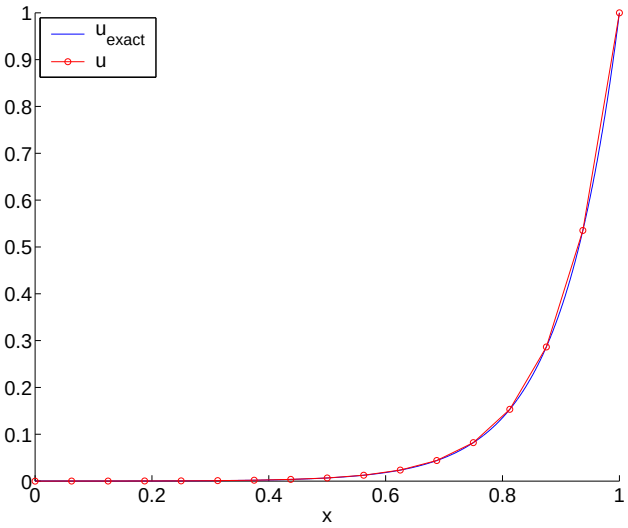


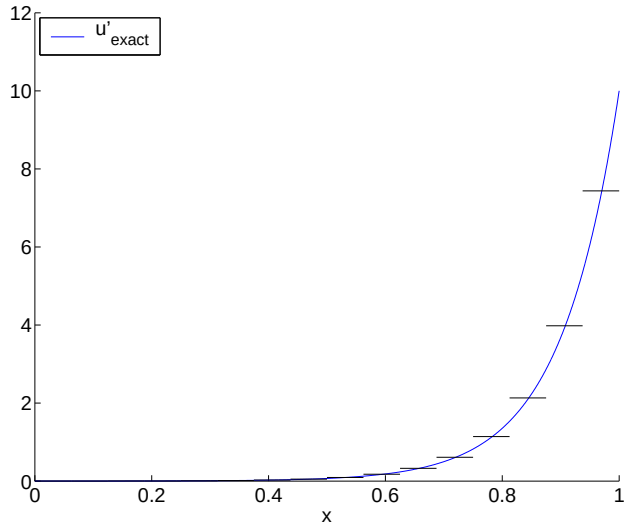


8 elements

$$||e||_{L_2} = 0.027$$

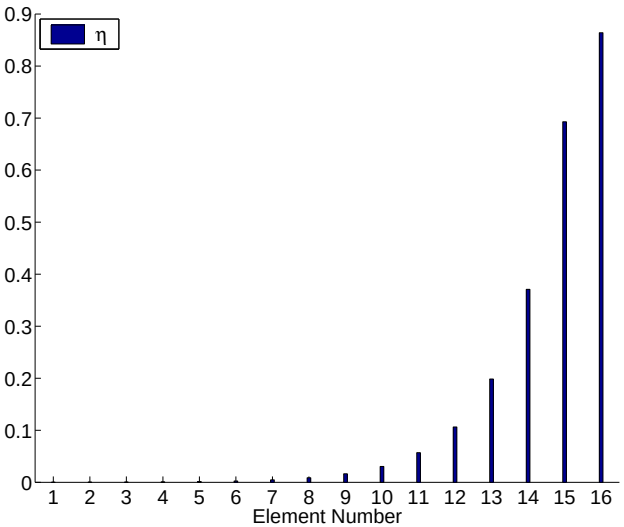






16 elements

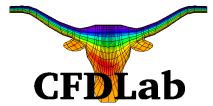
$$||e||_{L_2} = 0.0071$$

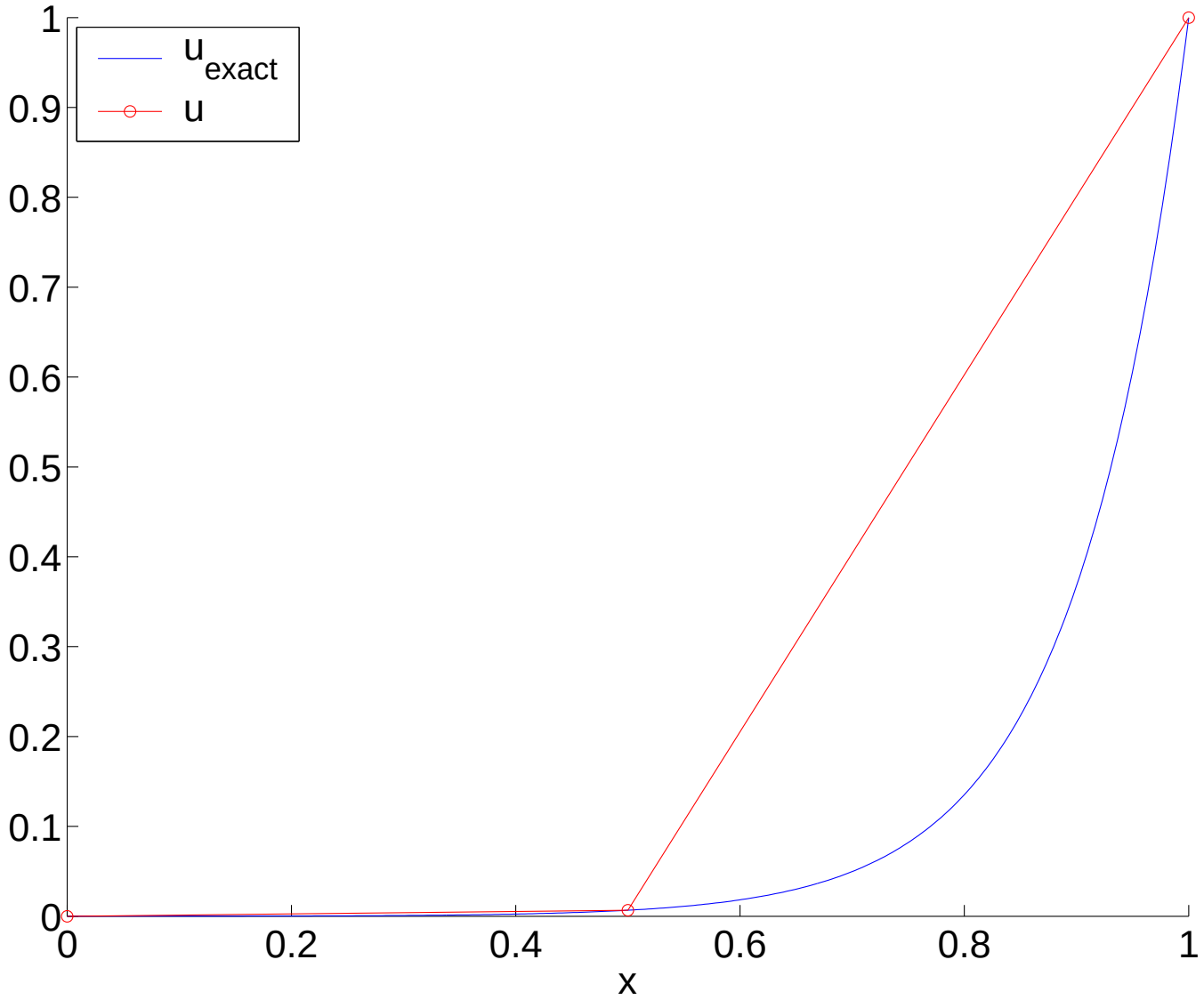


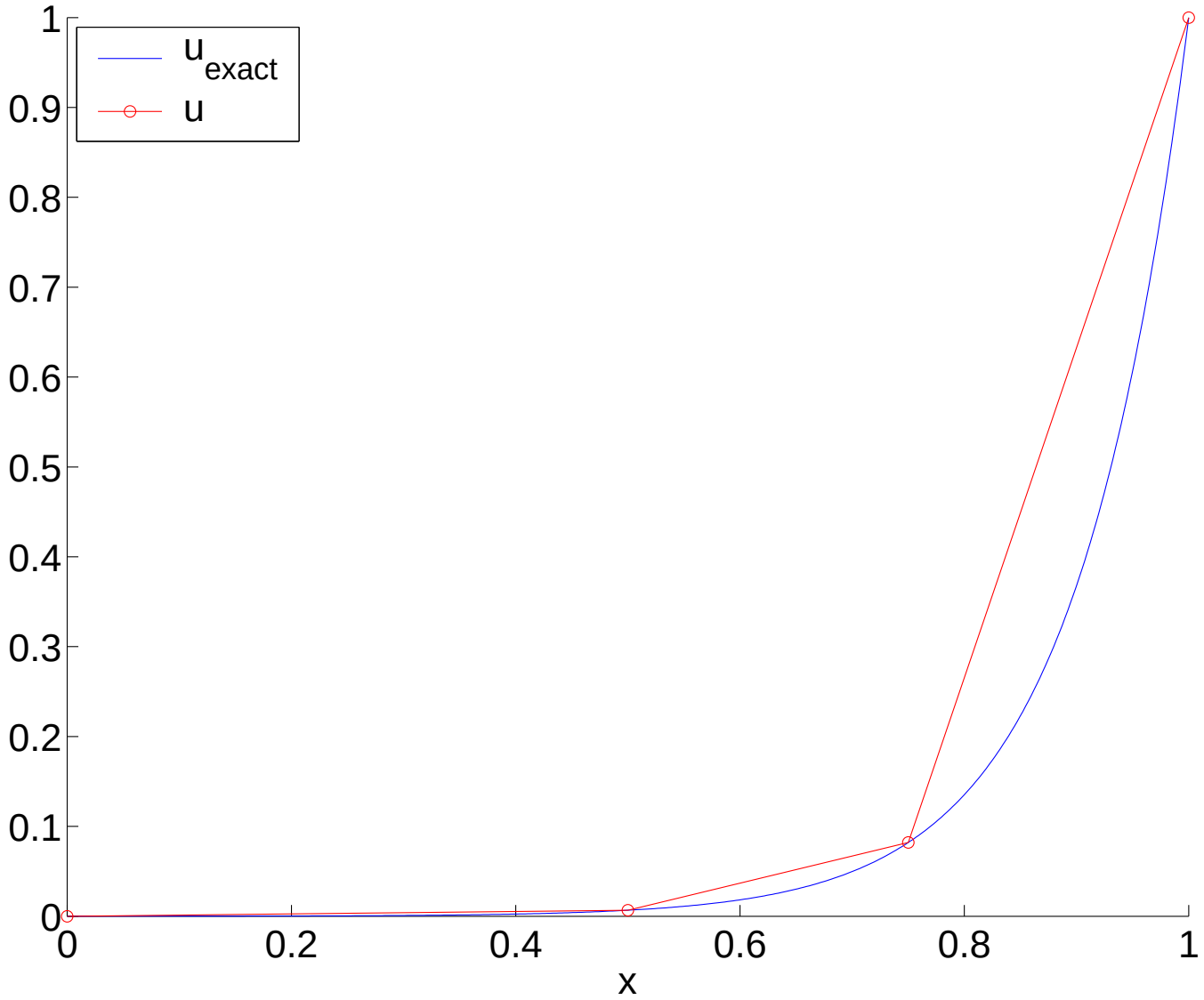
A Simple Refinement Strategy

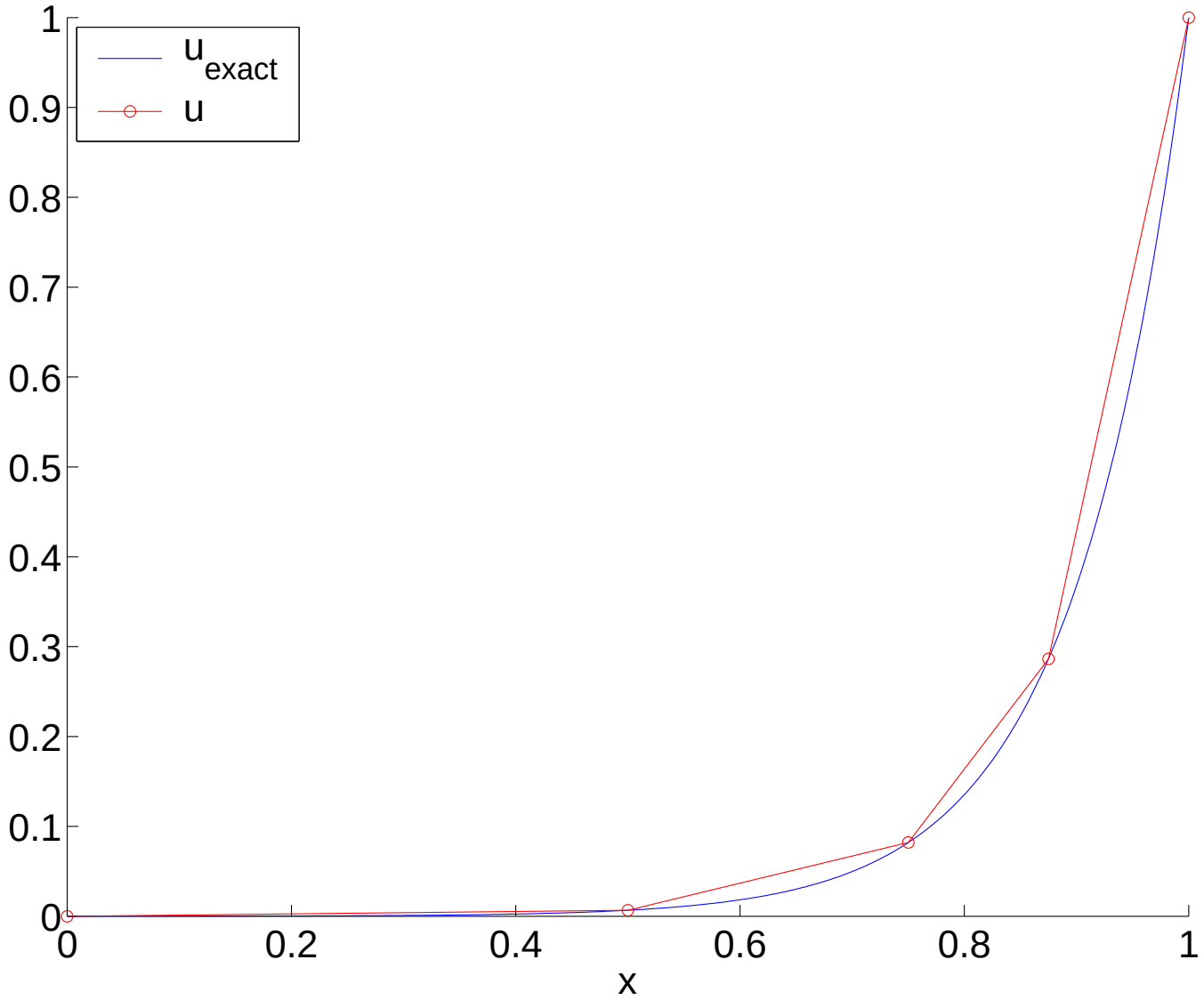
- A simple adaptive refinement strategy with `r_max` refinement steps for this 1D example problem is:

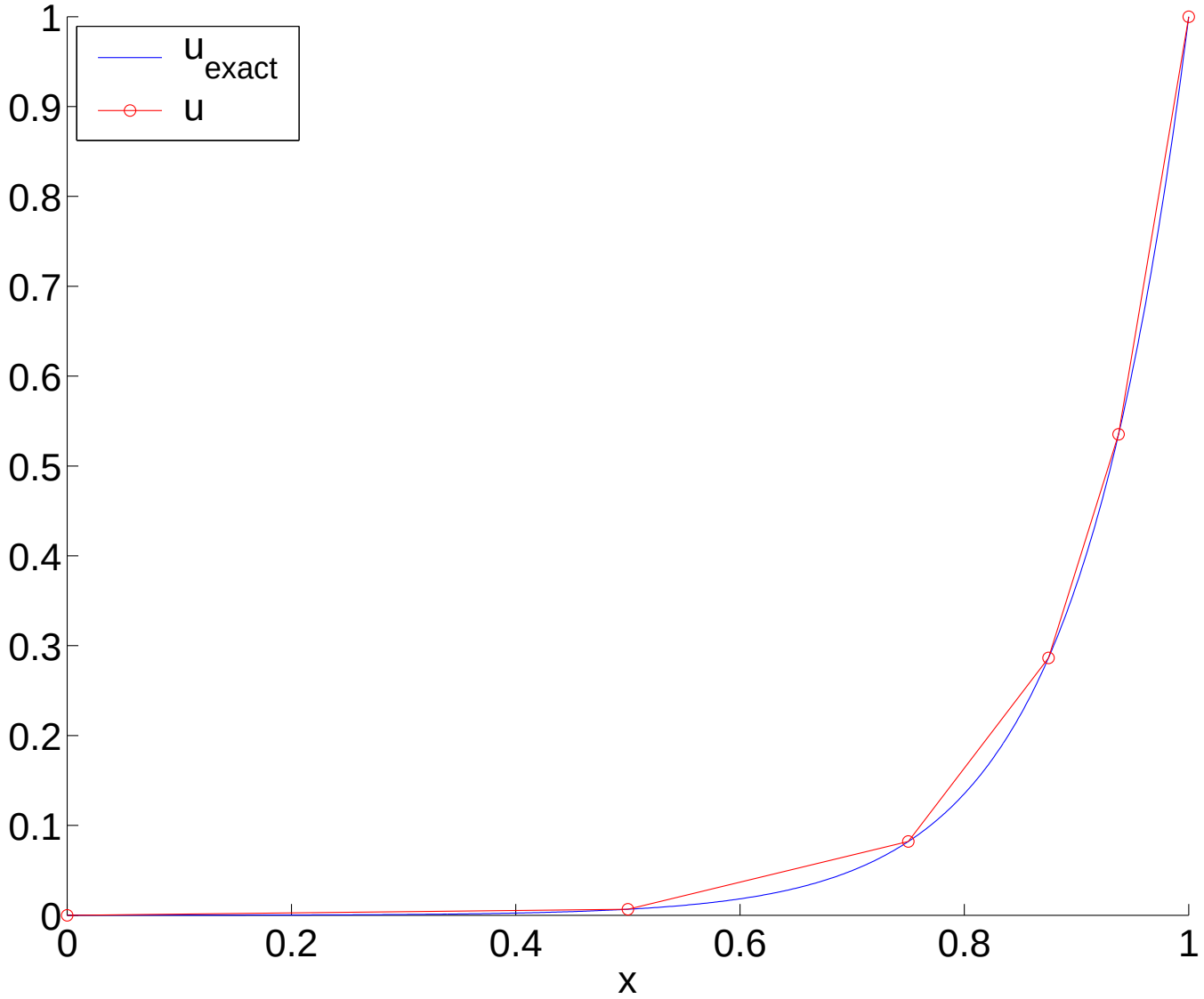
```
r=0;
while (r < r_max)
  Compute the FE solution (linear interpolant)
  Estimate the error (using flux-jump indicator)
  Refine the elements with error in top 10%
  Increment r
end
```

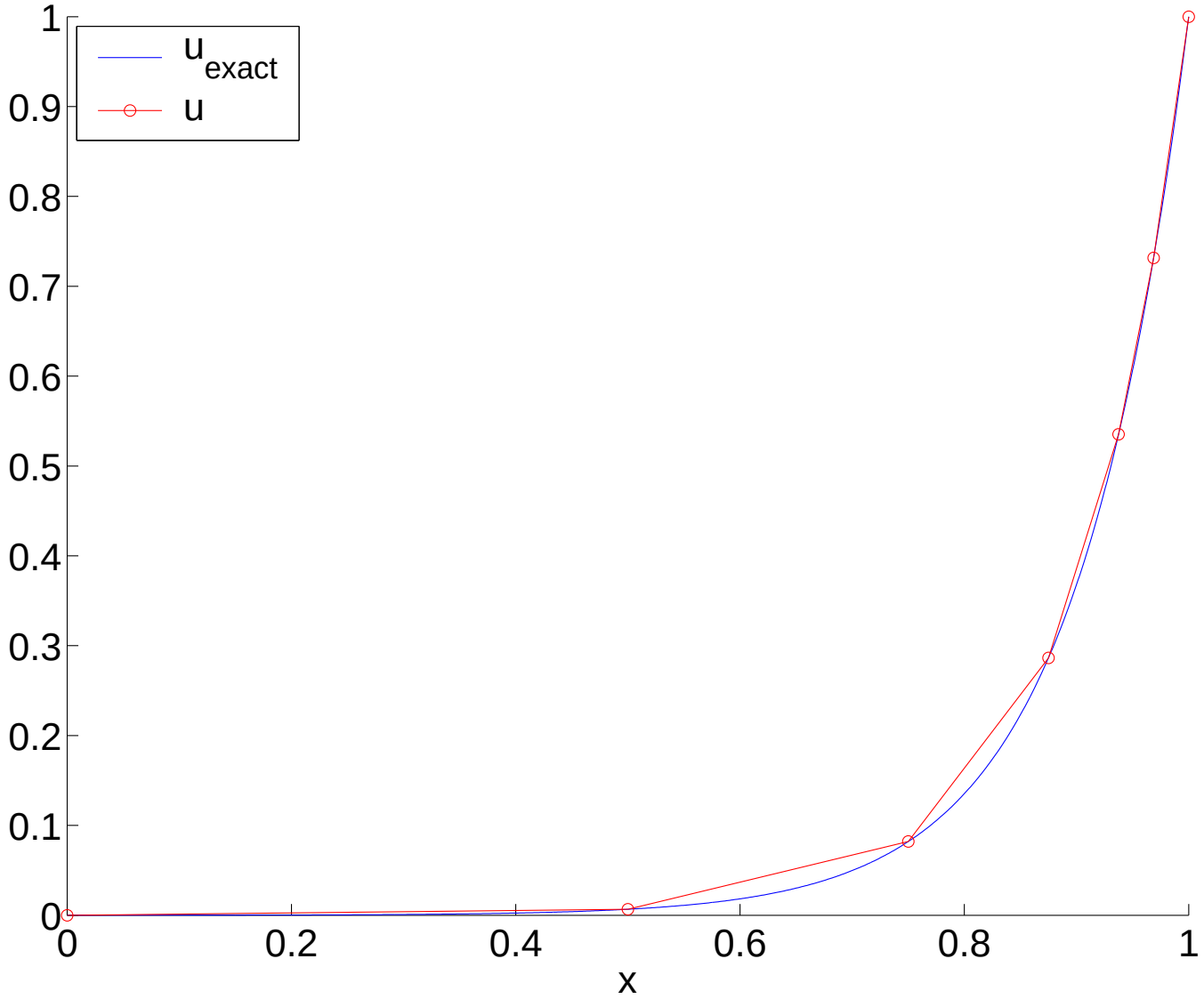


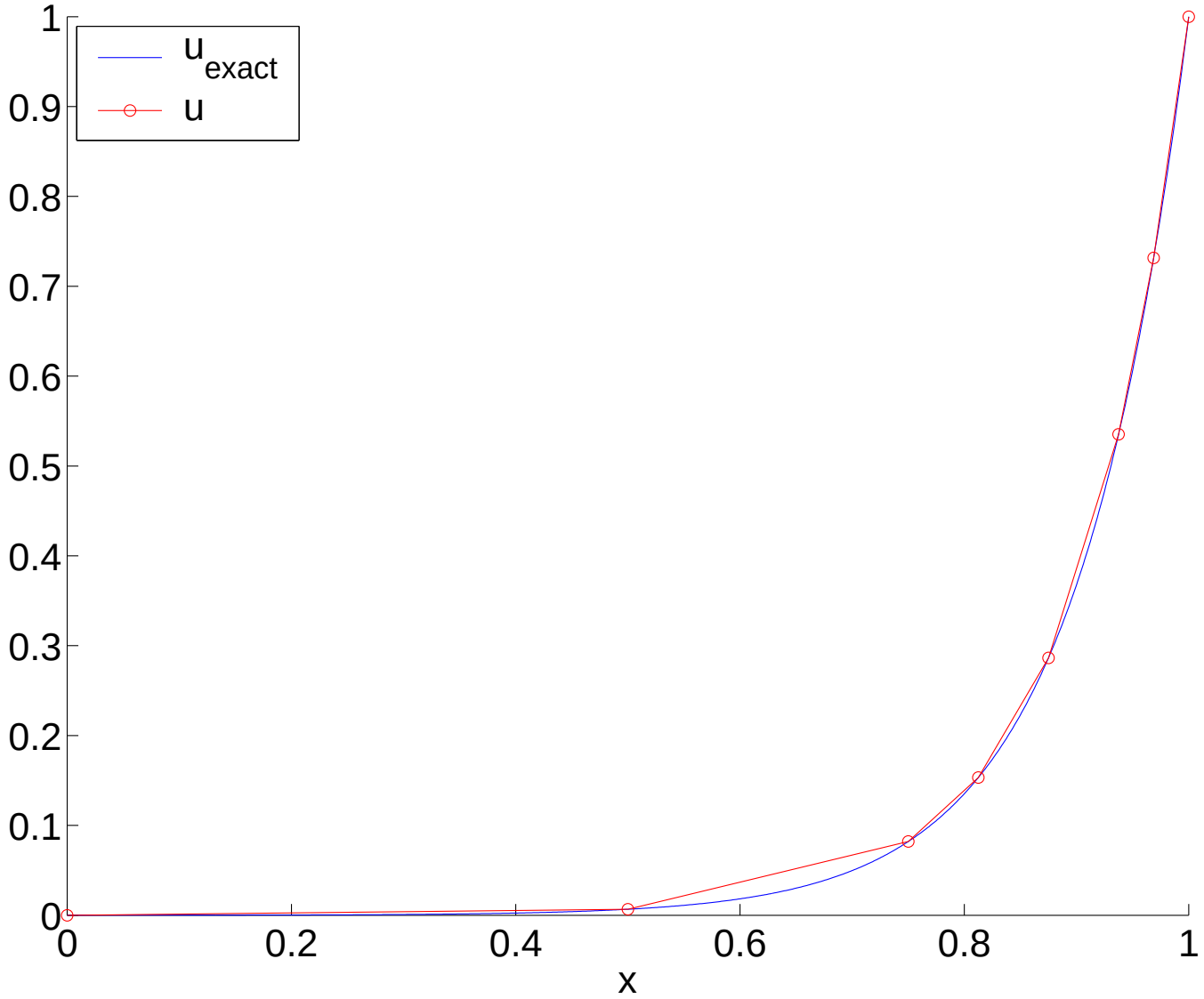

u_exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

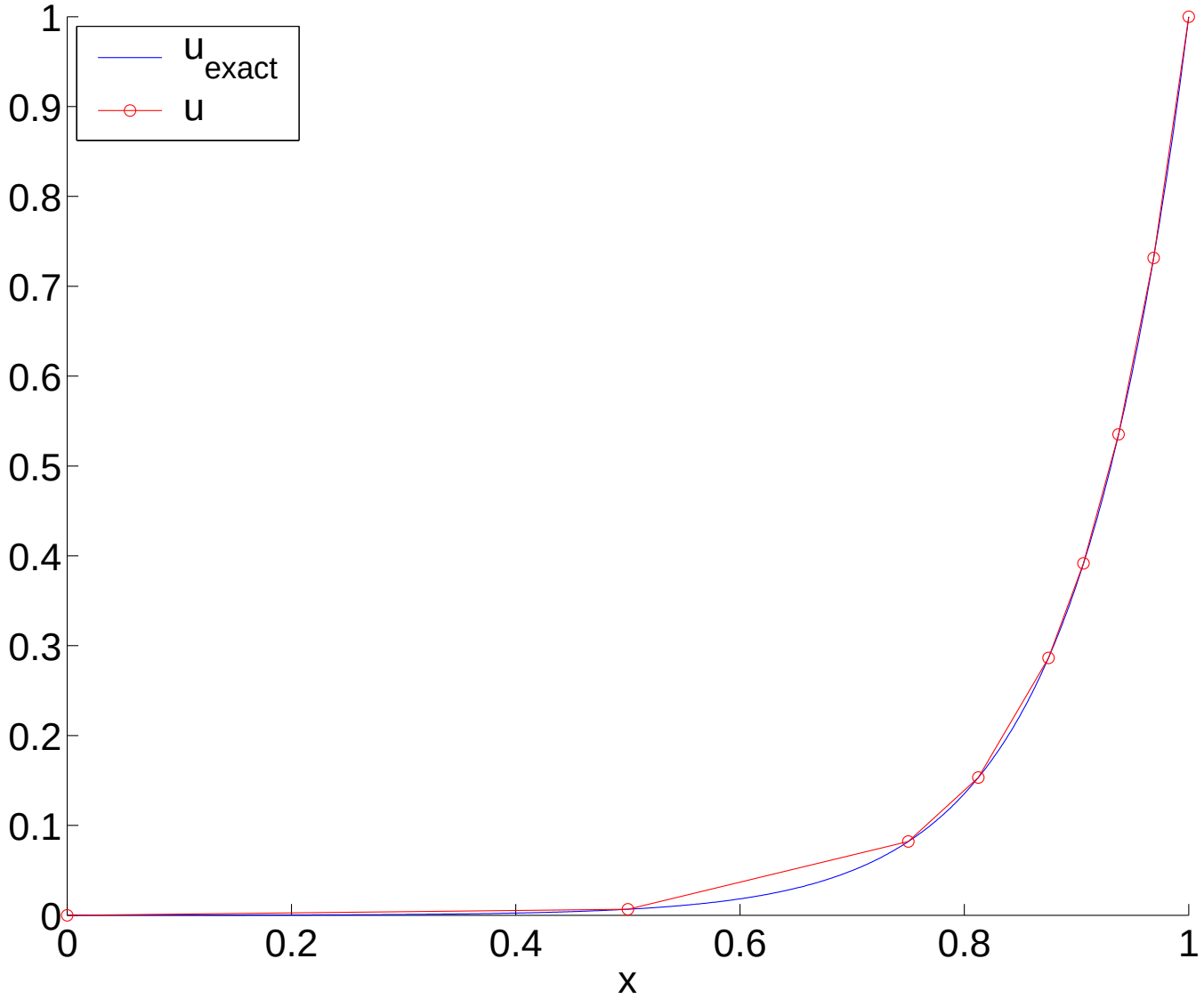

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

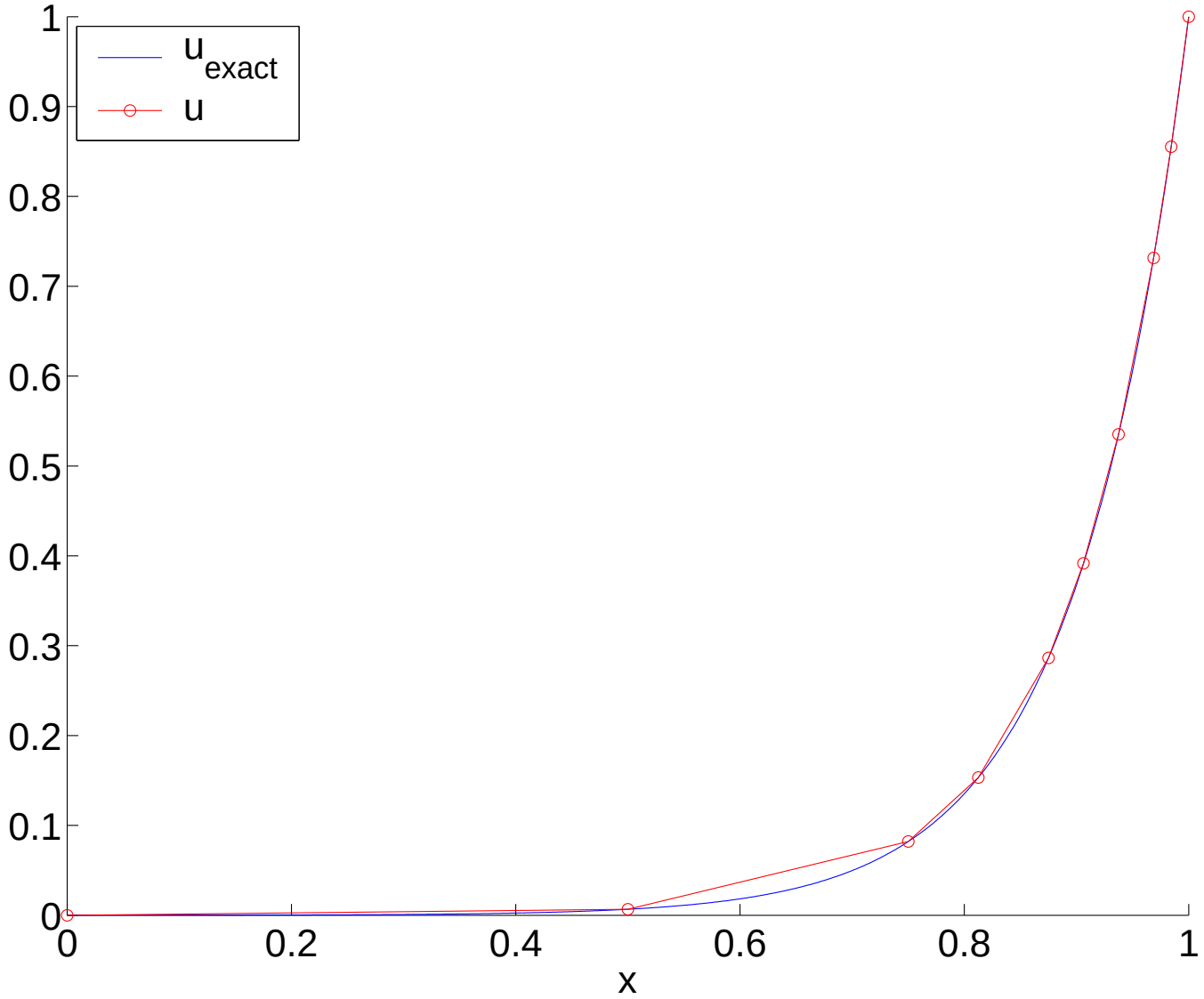

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

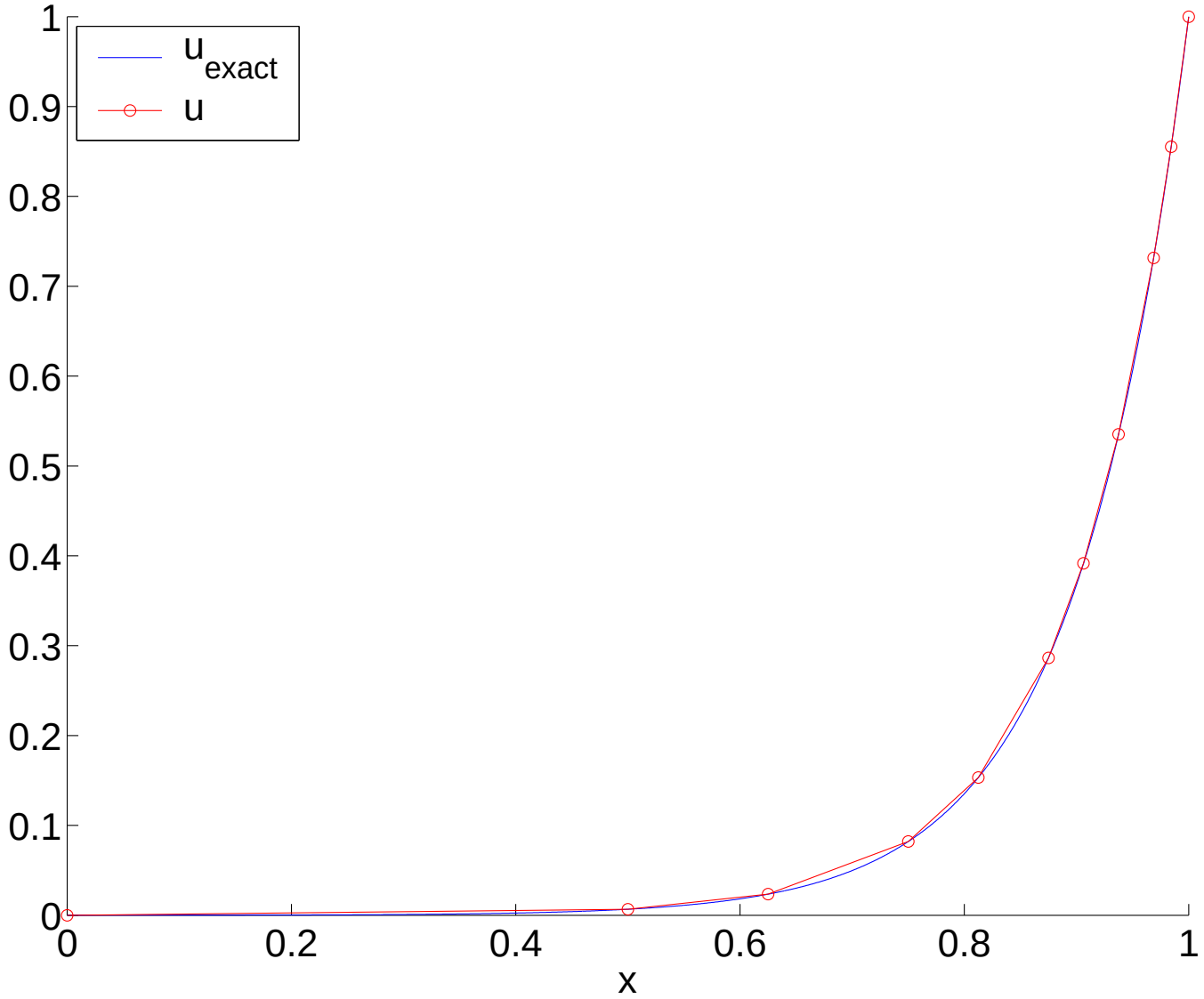

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

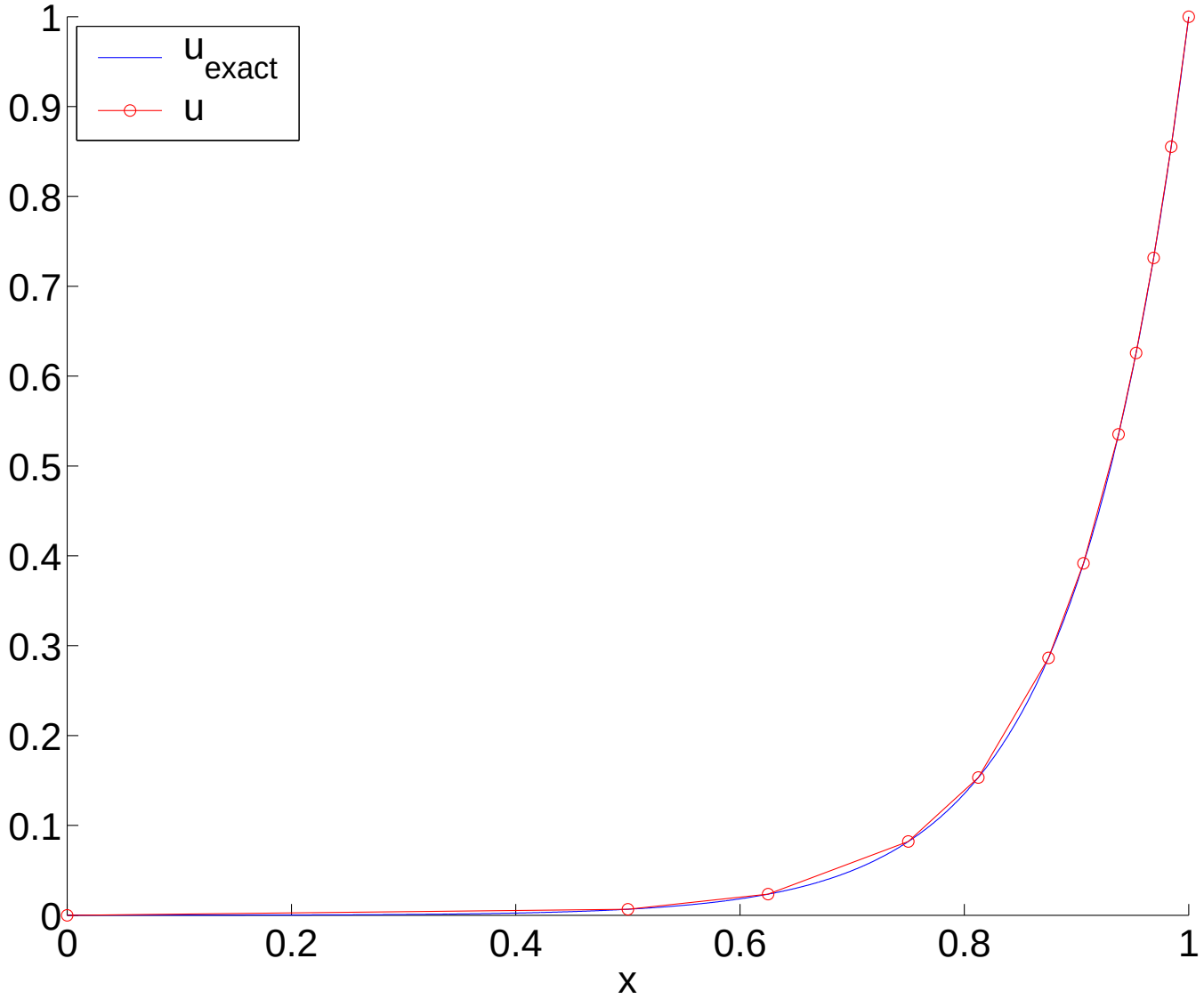

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

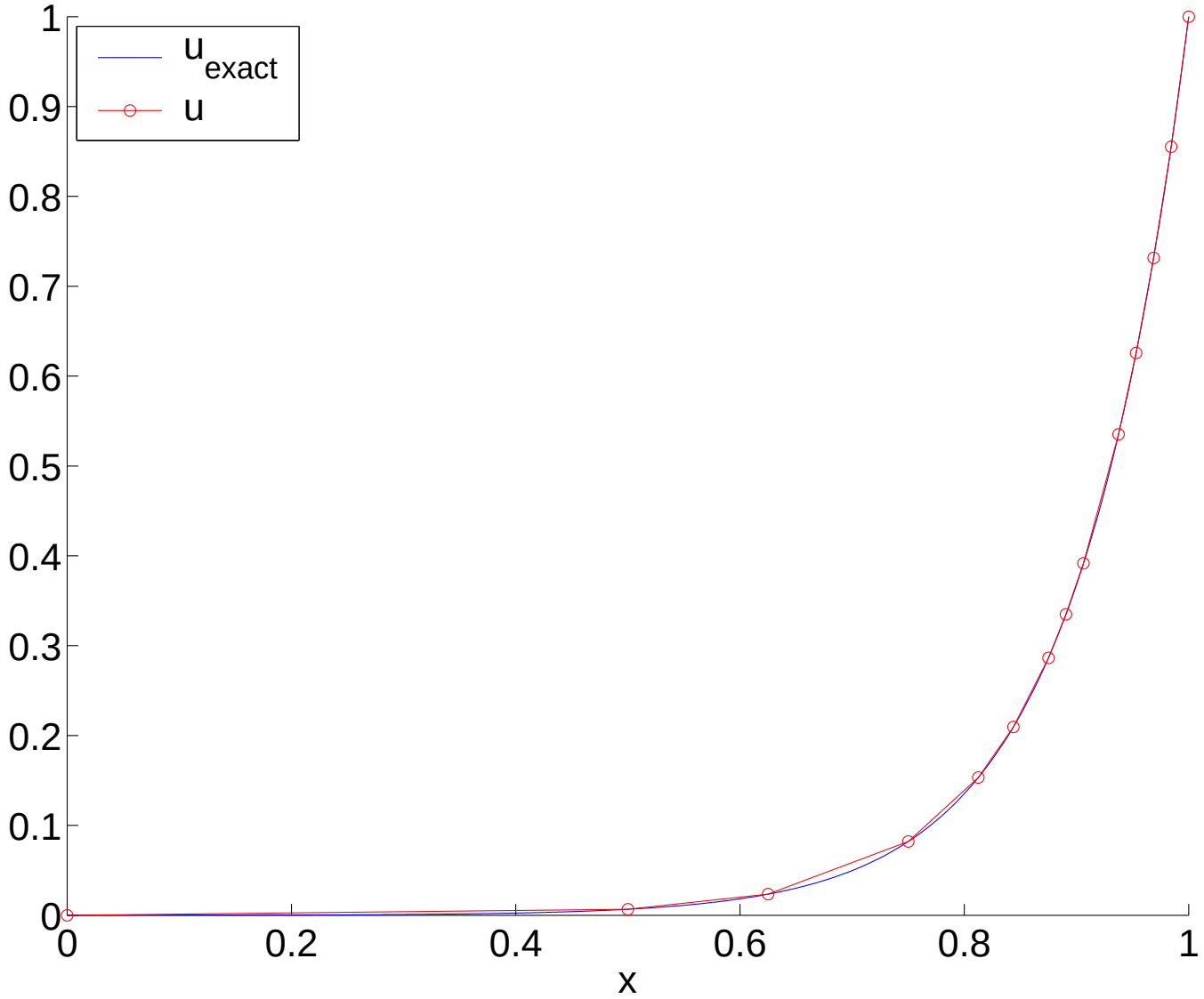

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x


u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

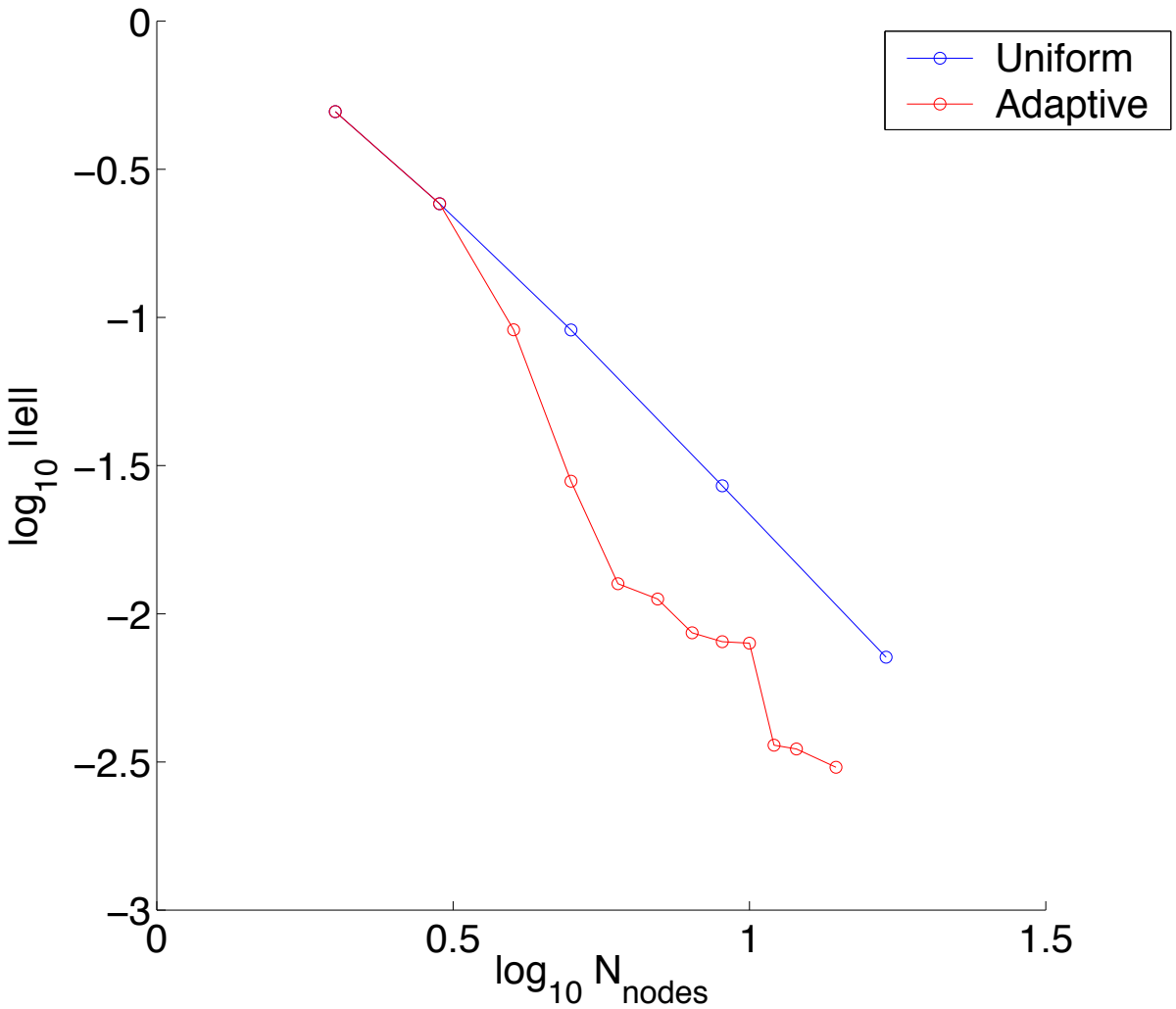

u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x


u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x


u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x


u exact
u
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
x

A Simple Refinement Strategy (cont.)

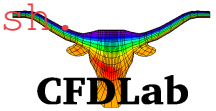


Examples: Transient Problem with AMR

`transient_convection_diffusion_AMR`

The `transient_convection_diffusion_AMR` program I

```
MeshRefinement mesh_refinement (mesh);
...

for (unsigned int r_step=0; r_step<max_r_steps; r_step++)
  {
    // Assemble & solve the linear system
    system.solve();

    // Print out the H1 norm, for verification purposes:
    Real H1norm = system.calculate_norm(*system.solution, SystemNorm(H1));
    std::cout << "H1 norm = " << H1norm << std::endl;

    // Possibly refine the mesh
    if (r_step+1 != max_r_steps)
      {
        std::cout << "  Refining the mesh..." << std::endl;

        // The ErrorVector is a particular StatisticsVector
        // for computing error information on a finite element mesh
        ErrorVector error;
```

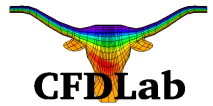


The `transient_convection_diffusion_AMR` program II

```
// The ErrorEstimator class interrogates a finite element
// solution and assigns to each element a positive error value.
// This value is used for deciding which elements to refine
// and which to coarsen.
KellyErrorEstimator error_estimator;

// Compute the error for each active element using the provided
// flux_jump indicator.  Note in general you will need to
// provide an error estimator specifically designed for your
// application.
error_estimator.estimate_error (system, error);

// This takes the error in error and decides which elements
// will be coarsened or refined.  Any element within 20% of the
// maximum error on any element will be refined, and any
// element within 7% of the minimum error on any element might
// be coarsened. Note that the elements flagged for refinement
// will be refined, but those flagged for coarsening _might_ be
// coarsened.
```

The `transient_convection_diffusion_AMR` program III

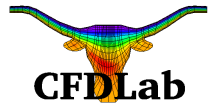
```
      mesh_refinement.refine_fraction() = 0.80;
      mesh_refinement.coarsen_fraction() = 0.07;
      mesh_refinement.max_h_level() = 5;
      mesh_refinement.flag_elements_by_error_fraction (error);
      mesh_refinement.refine_and_coarsen_elements();

      // This call reinitializes the EquationSystems object for
      // the newly refined mesh.  One of the steps in the
      // reinitialization is projecting the solution,
      // old_solution, etc... vectors from the old mesh to
      // the current one.
      equation_systems.reinit ();
    }
}
```



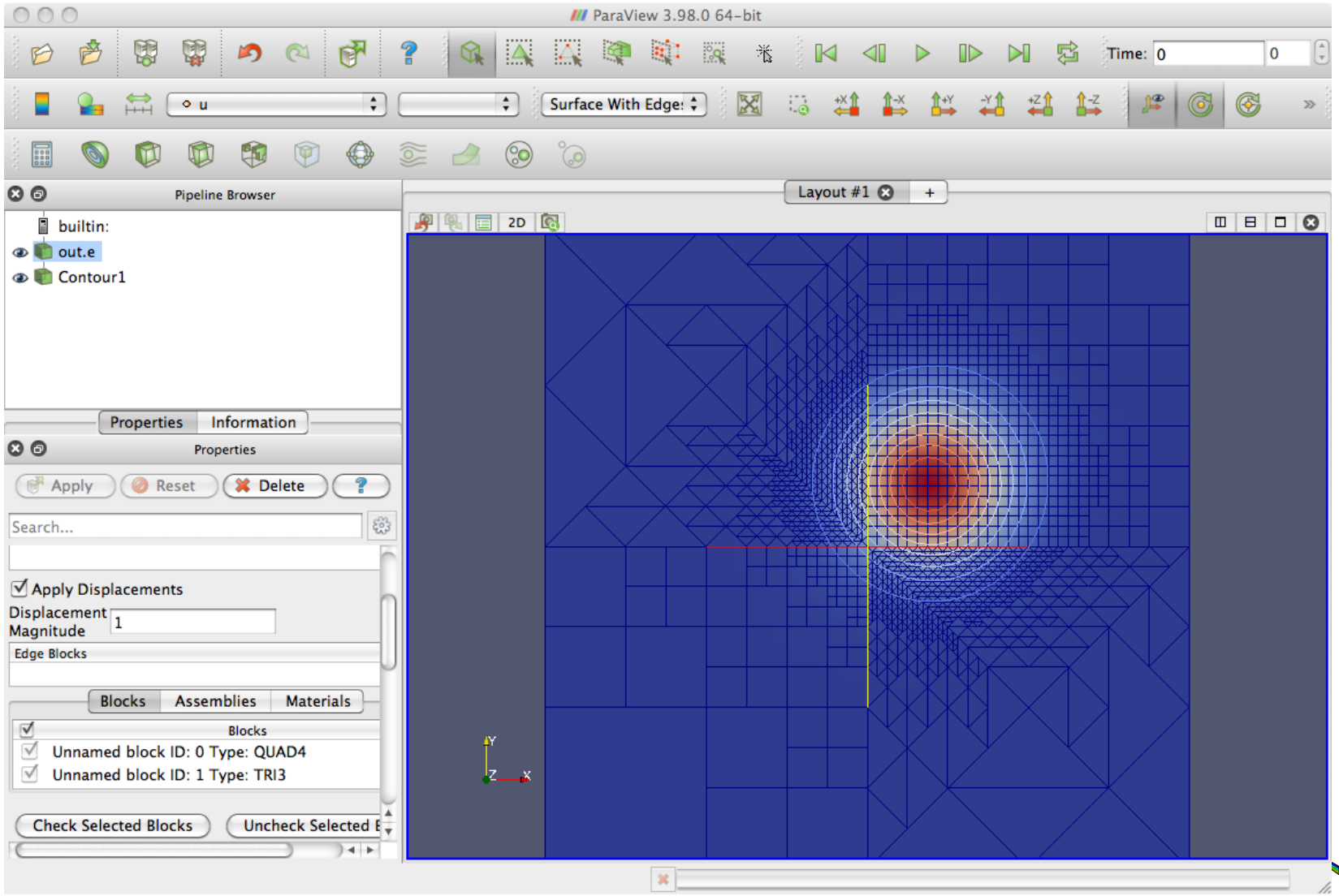
Running the transient_convection_diffusion_AMR program

Running the program

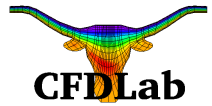
```
# copy the example
$ cp -r $LIBMESH_TUTORIAL/transient_convection_diffusion_AMR .
$ cd transient_convection_diffusion_AMR
$ make

# run the example for 25 timesteps from an initially refined mesh
$ ./example-opt -n_timesteps 25 -n_refinements 5 \
                -output_freq 10 -init_timestep 0

# restart the example, reading the refined mesh and solution
$ ./example-opt -read_solution -n_timesteps 25 \
                -output_freq 10 -init_timestep 25
```



Output

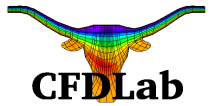


CFDLab



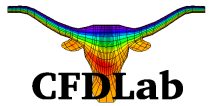
Parallel Data Structures in `libMesh`

`ParallelMesh`



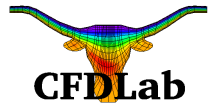
FEM Computational Costs

- Discrete system solves
- Discrete Jacobian assembly
- Discrete residual assembly
- Sparse Jacobian allocation
- I/O
- Mesh generation
- Mesh movement



Adaptive FEM Costs

- Error estimator evaluation
- Adaptive refinement/coarsening
- Inter-mesh projections
- Adaptivity flagging
- Adaptive constraint calculations



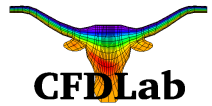
Parallelism Goals

Reduced CPU time

- Distributed CPU usage
- Asynchronous I/O

Reduced memory requirements

- Larger attainable problem size



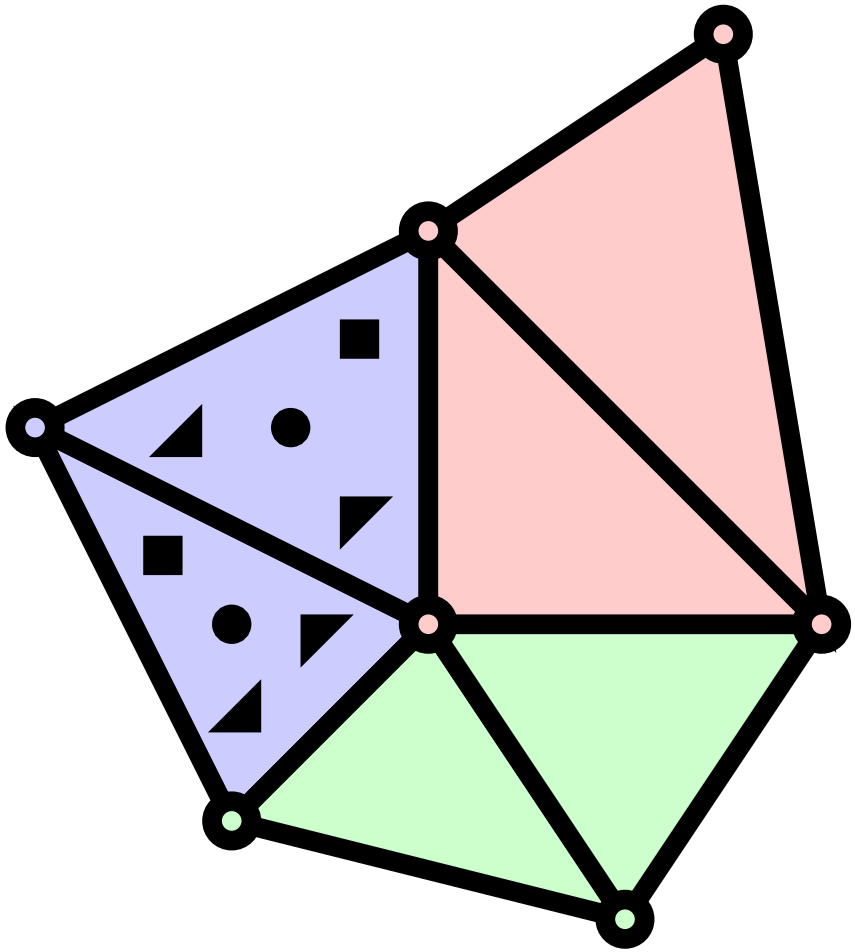
Parallelism Levels

SIMD Instructions

- Assemblies, MatVec operations

Shared-memory Threads

- Mesh Partitioning



CFDLab

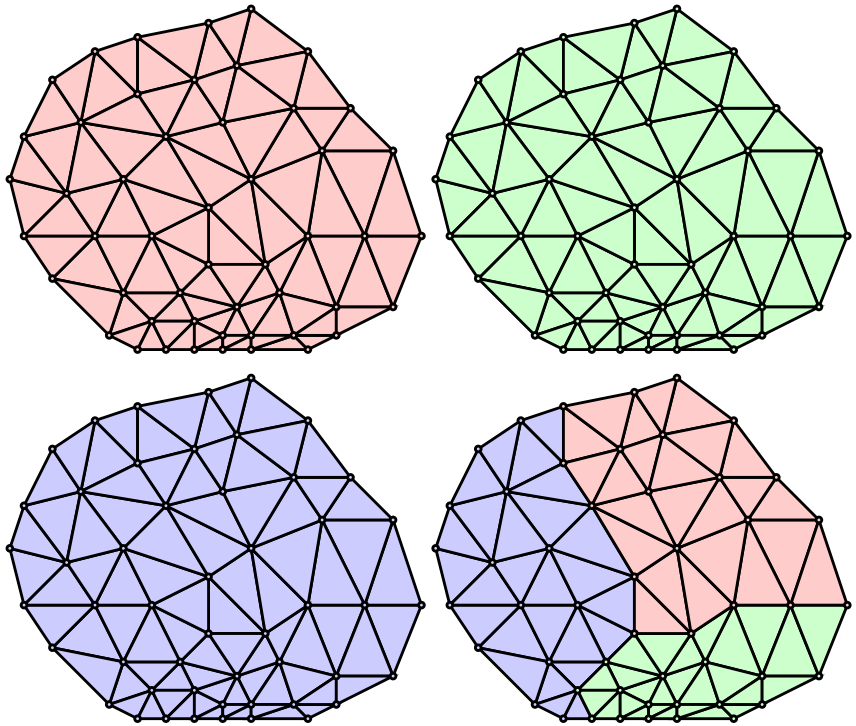
Parallelism Levels

Separate Simulations

- Parametric studies
- Uncertainty analysis

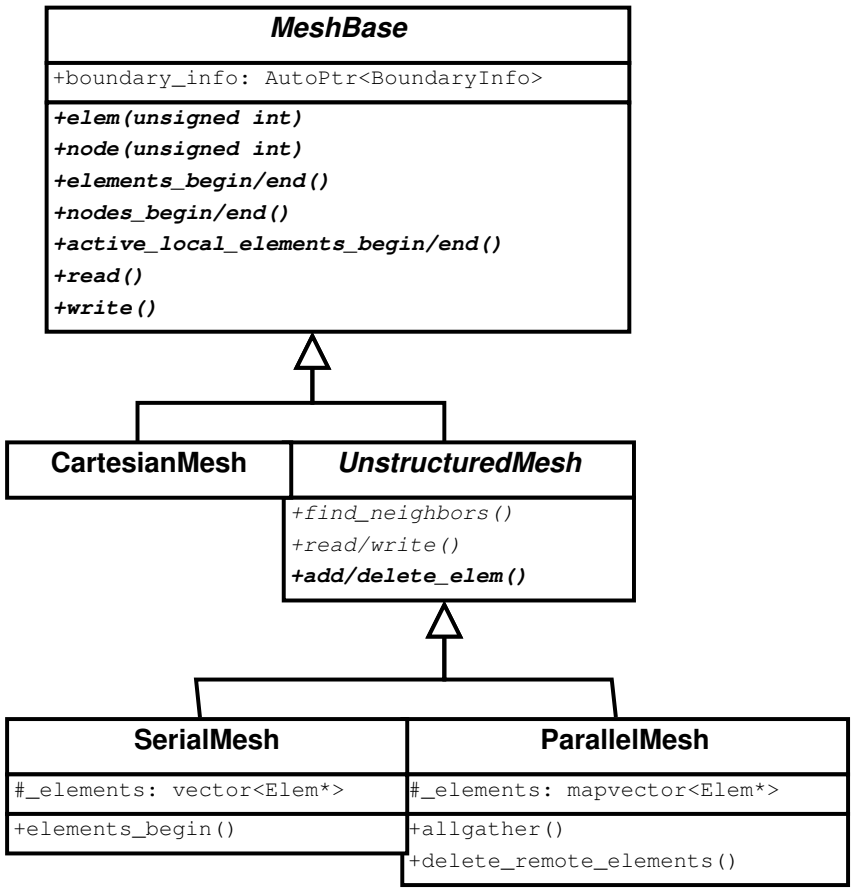
Distributed-memory Processes

- Asynchronous I/O
- Mesh Partitioning



CFDLab

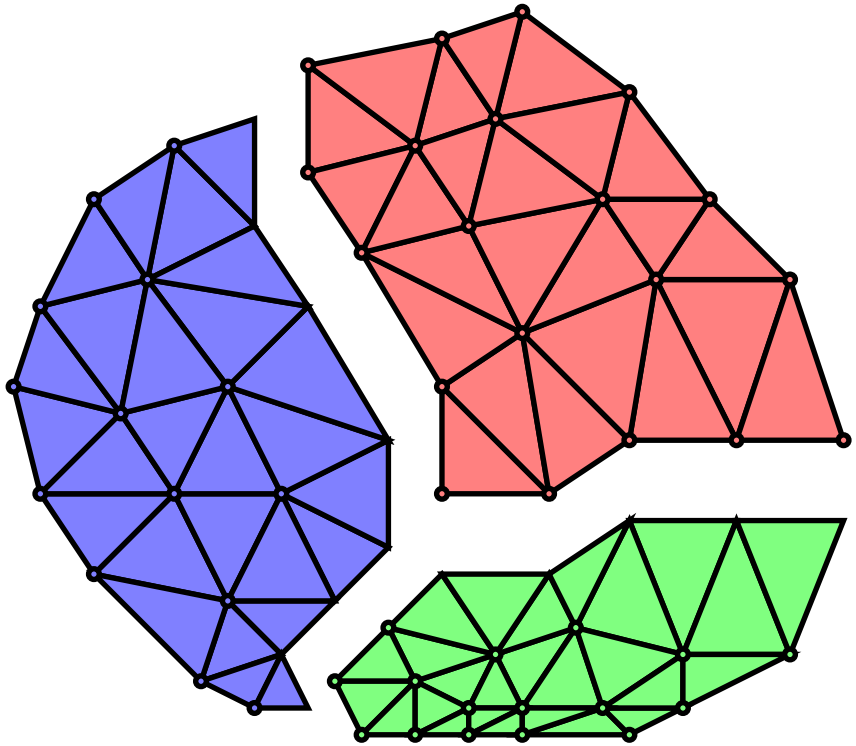
Mesh Classes



- Abstract iterator interface hides mesh type from most applications
- UnstructuredMesh ”branch” for most library code
- ParallelMesh implements data storage, synchronization

SerialMesh Partitioning

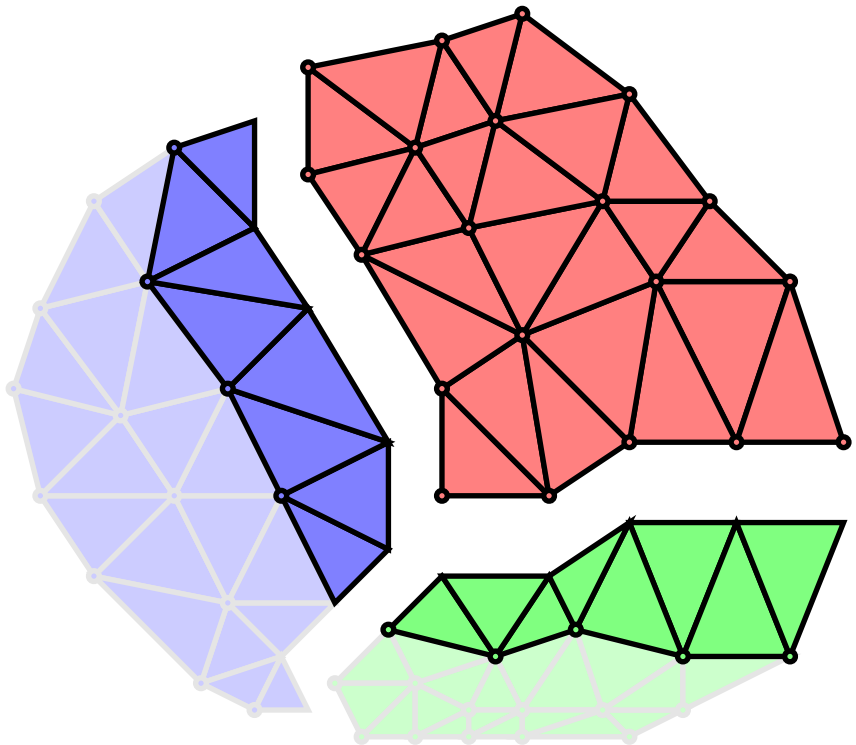
- Each element, node is “local” to one processor
- Each processor has an identical Mesh copy
- Mesh stays in sync through redundant work
- FEM data synced on “ghost” elements only



CFDLab

ParallelMesh Partitioning

- Processors store only local and ghost objects
- Each processor has a small Mesh subset
- Mesh stays in sync through MPI communication

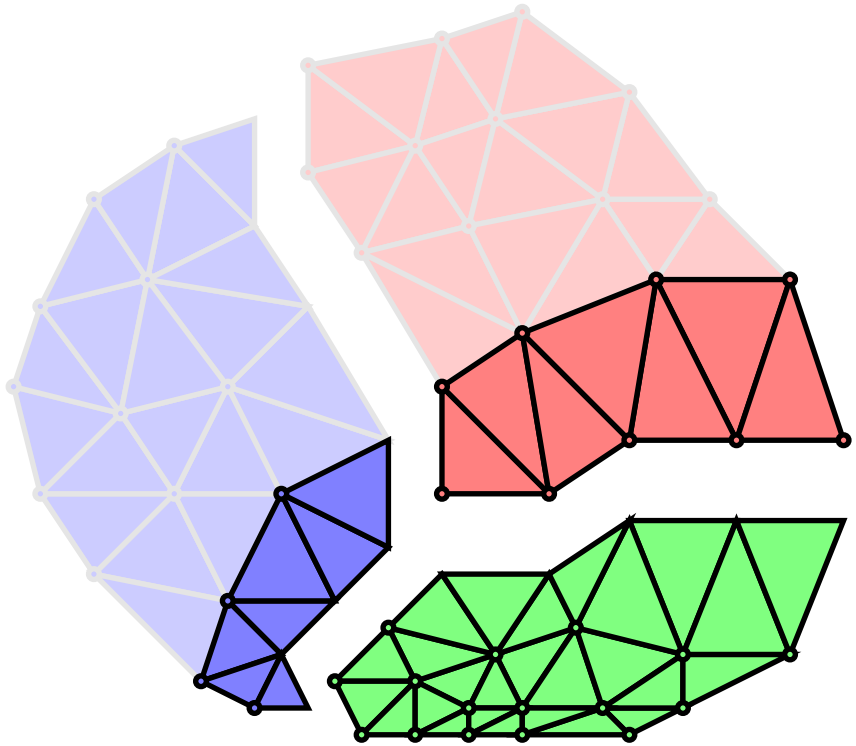


CFDLab

ParallelMesh Partitioning

Pros

- Reduced memory use
- $O(N_E/N_P)$ CPU costs

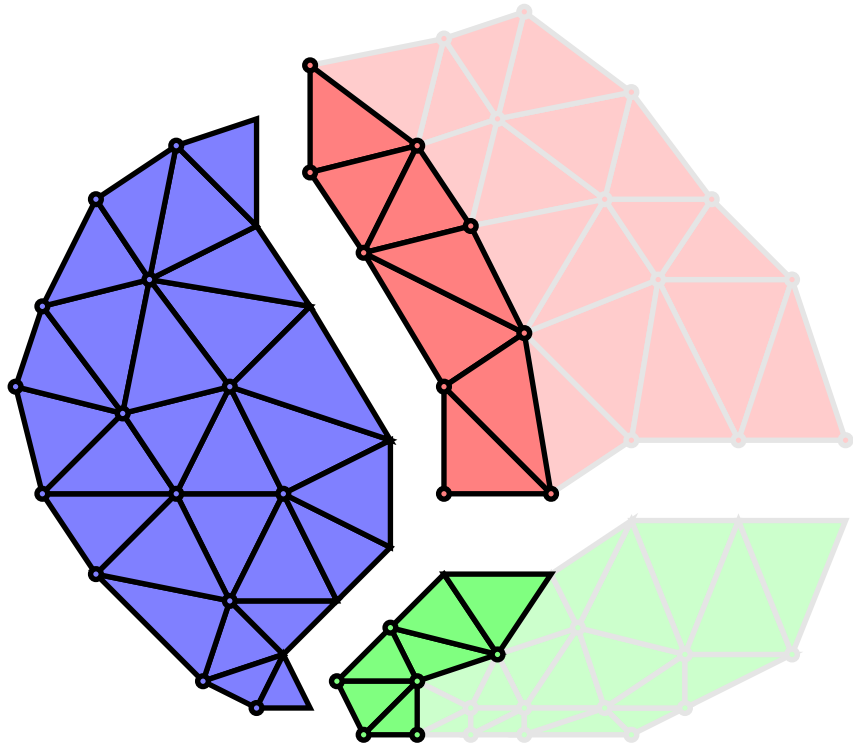


CFDLab

ParallelMesh Partitioning

Cons

- Increased code complexity
- Increased synchronization "bookkeeping"
- Greater debugging difficulty



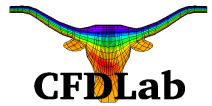
CFDLab

Gradual Parallelization

Starting from SerialMesh behavior

- New internal data structure
- Methods to delete, reconstruct non-semilocal objects
- Parallelized DofMap methods
- Parallelized MeshRefinement methods
- Parallel Mesh I/O support
- Load balancing support

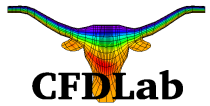
Also working on parallel support in Boundary, Function, Generation, Modification, Generation, Tools classes



Distributed Mesh Refinement

Error Estimation

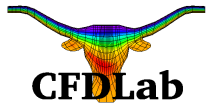
- Local residual, jump error estimators
- Refinement-based estimators
- Adjoint-based estimators
- Recovery estimators



Distributed Mesh Refinement

Error Estimation

- Local residual, jump error estimators: embarrassingly parallel
- Refinement-based estimators: use solver parallelism
- Adjoint-based estimators: use solver parallelism
- Recovery estimators: require partitioning-aware patch generation



Distributed Mesh Refinement

Error Estimation

- Local residual, jump error estimators: embarrassingly parallel
- Refinement-based estimators: use solver parallelism
- Adjoint-based estimators: use solver parallelism
- Recovery estimators: require partitioning-aware patch generation

Refinement Flagging

- Flagging by error tolerance: $\eta_K^2 < \eta_{tol}^2 / N_E$
- Flagging by error fraction: $\eta_K < r \max_K \eta_K$
- Flagging by element fraction or target N_E

Distributed Mesh Refinement

Error Estimation

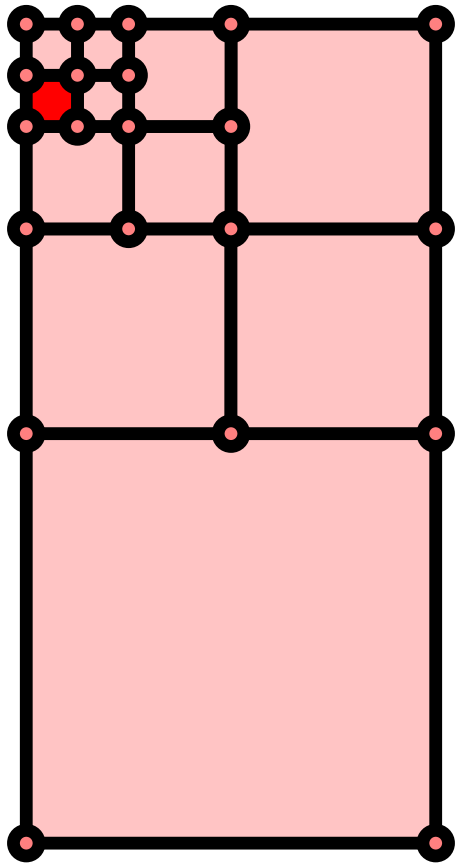
- Local residual, jump error estimators: embarrassingly parallel
- Refinement-based estimators: use solver parallelism
- Adjoint-based estimators: use solver parallelism
- Recovery estimators: require partitioning-aware patch generation

Refinement Flagging

- Flagging by error tolerance: $\eta_K^2 < \eta_{tol}^2/N_E$
  - Embarrassingly parallel
- Flagging by error fraction: $\eta_K < r \max_K \eta_K$
  - One global Parallel::max to find maximum error
- Flagging by element fraction or target N_E
  - Parallel::Sort to find percentile levels?
  - Binary search in parallel?
  - TBD

Distributed Mesh Refinement

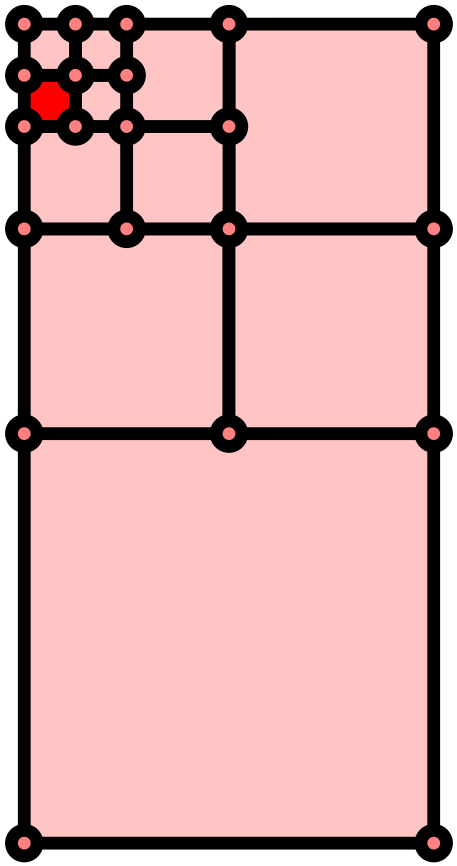
Elem, Node creation



Distributed Mesh Refinement

Elem, Node creation

- Ids $\{i : i \bmod (N_P + 1) = p\}$ are owned by processor p



CFDLab

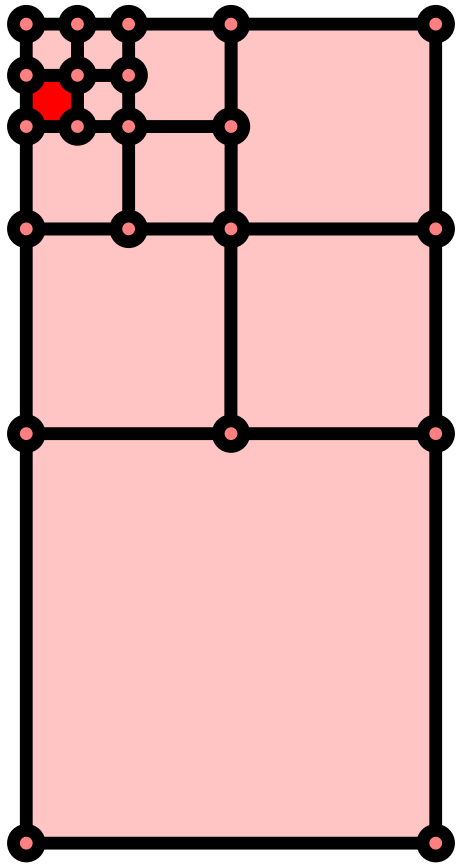
Distributed Mesh Refinement

Elem, Node creation

- Ids $\{i : i \bmod (N_P + 1) = p\}$ are owned by processor p

Synchronization

- Refinement Flags
- New ghost child elements, nodes
- Hanging node constraint equations



CFDLab

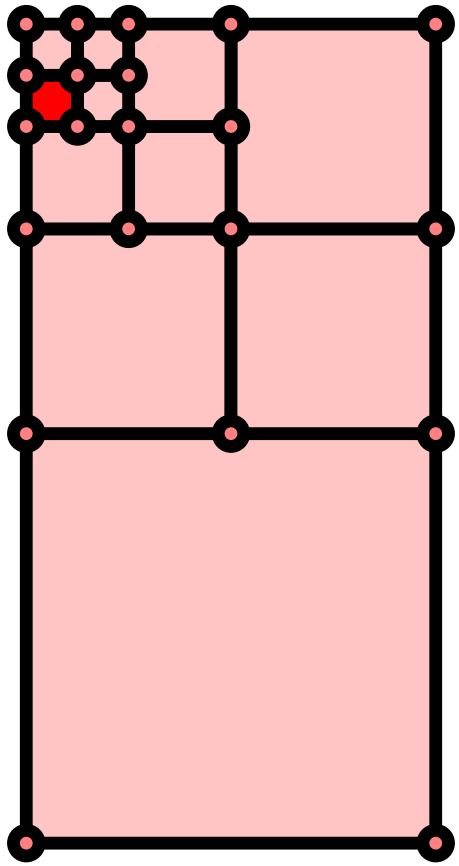
Distributed Mesh Refinement

Elem, Node creation

- Ids $\{i : i \bmod (N_P + 1) = p\}$ are owned by processor p

Synchronization

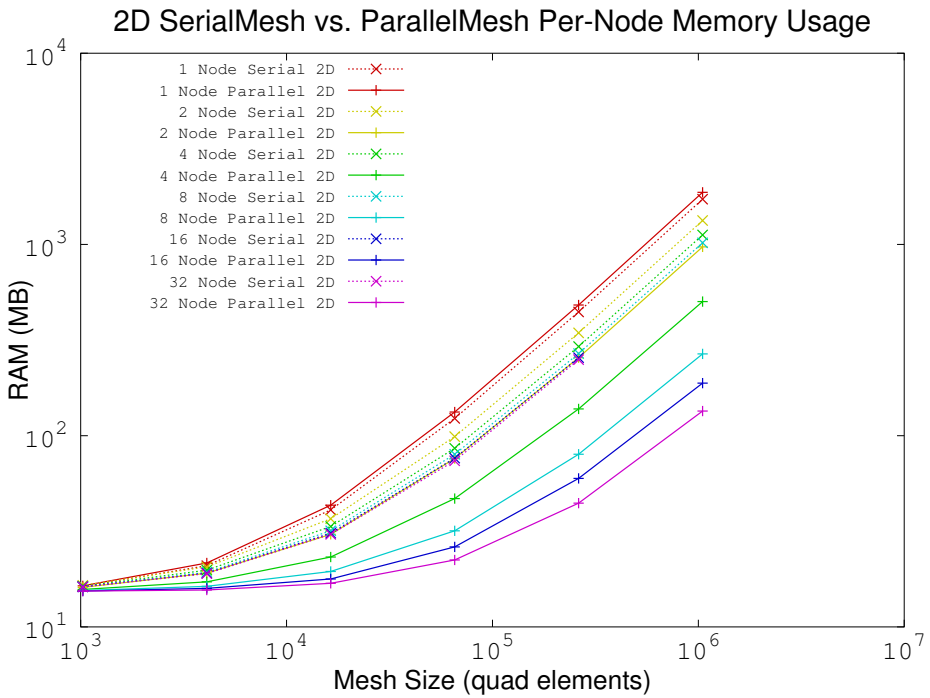
- Refinement Flags
  - Data requested by id
  - Iteratively to enforce smoothing
- New ghost child elements, nodes
  - Id requested by data
- Hanging node constraint equations
  - Iteratively through subconstraints, subconstraints-of-subconstraints...

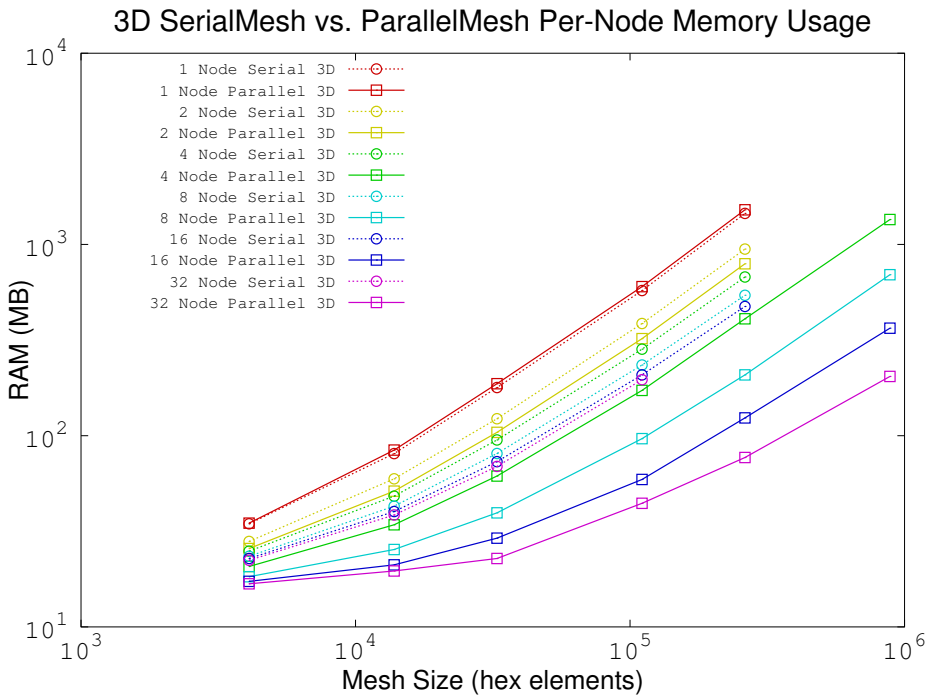


CFDLab

“Typical” PDE example

Transient Cahn-Hilliard, Bogner-Fox-Schmidt quads or hexes





Results

- Parallel codes using SerialMesh are unchanged for ParallelMesh
- Overhead, distributed sparse matrix costs are unchanged
- Serialized mesh, indexing once dominated RAM use

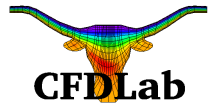
Parallel:: API

Encapsulating MPI

- Improvement over MPI C++ interface
- Makes code shorter, more legible

Example:

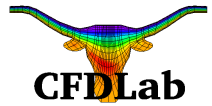
```
std::vector<Real> send, recv;
...
send_receive(dest_processor_id, send,
             source_processor_id, recv);
```



Parallel:: API

Instead of:

```
if (dest_processor_id   == libMesh::processor_id() &&
    source_processor_id == libMesh::processor_id())
  recv = send;
#ifdef HAVE_MPI
else
  {
    unsigned int sendsize = send.size(), recvsize;
    MPI_Status status;
    MPI_Sendrecv(&sendsize, 1, datatype<unsigned int>(),
                 dest_processor_id, 0,
                 &recvsize, 1, datatype<unsigned int>(),
                 source_processor_id, 0,
                 libMesh::COMM_WORLD,
                 &status);

    recv.resize(recvsize);

    MPI_Sendrecv(sendsize ? &send[0] : NULL, sendsize, MPI_DOUBLE,
                 dest_processor_id, 0,
                 recvsize ? &recv[0] : NULL, recvsize, MPI_DOUBLE,
                 source_processor_id, 0,
                 libMesh::COMM_WORLD,
                 &status);
  }
#endif // HAVE_MPI
```



ParallelMesh Data Structure

std::vector fails

- Not sparse
- $O(N_E)$ storage cost

std::map

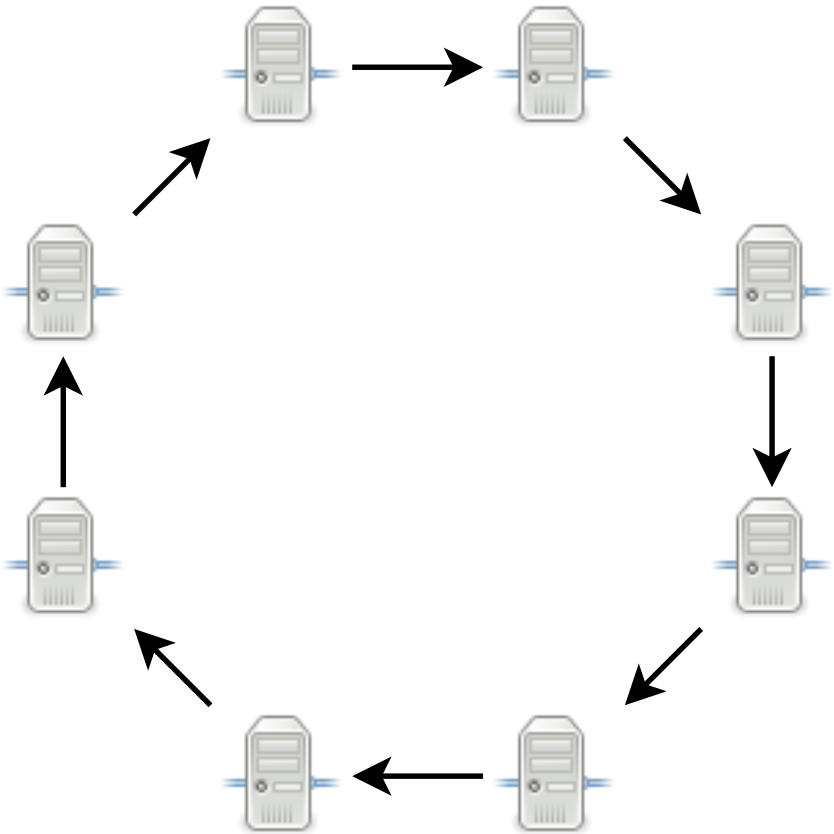
- "mapvector" interface provides iterators
- $O(log(N_E/N_P))$ lookup time without std::hash_map
- $O(1)$ lookup time with std::hash_map

Hybrid data structure?

- Dense vector for most elements
- Sparse data structure for new elements

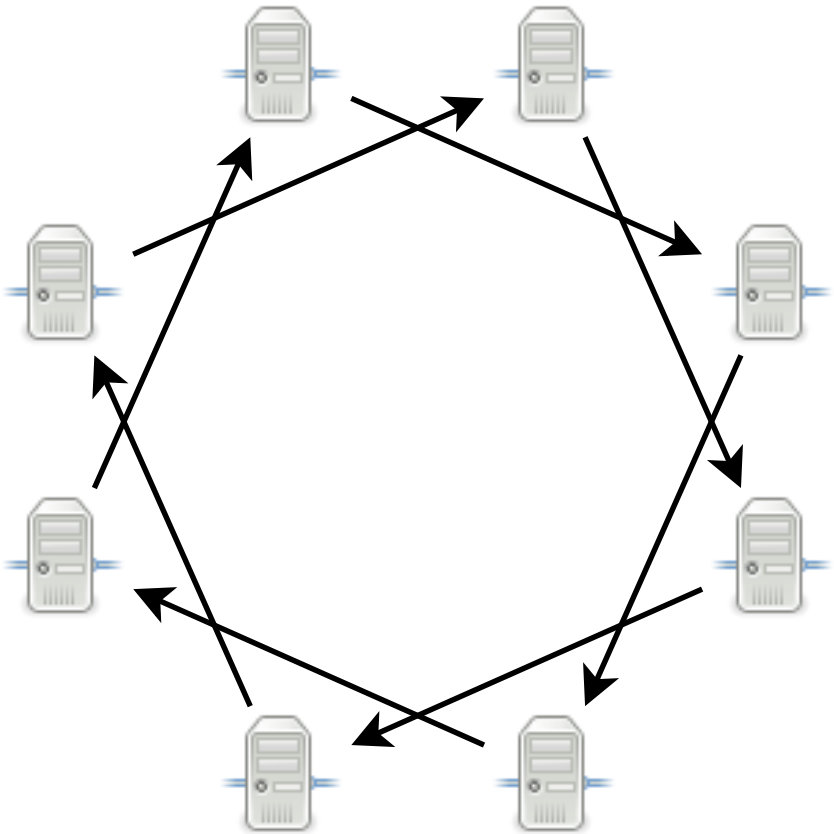
CFDLab

Round Robin Communications



- Processor P sends to processor $P+K$ while receiving from $P-K$
- New data is operated on and old data discarded
- K is incremented "round robin" from 1 to $N_P - 1$

Round Robin Communications



Pros

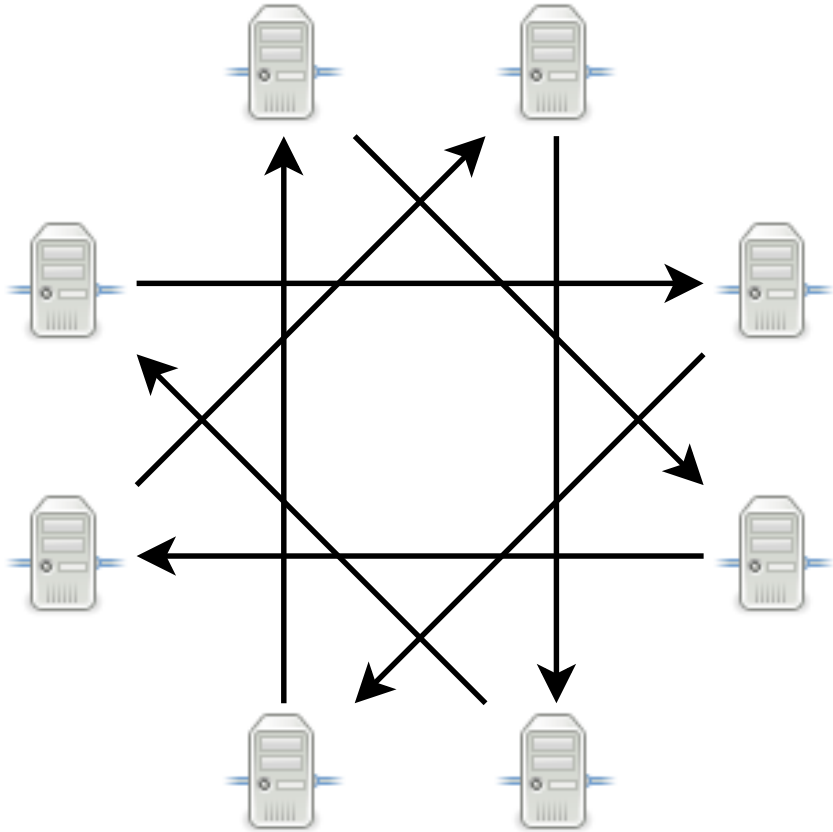
- $O(N_G/N_P)$ memory usage - only one data exchange at a time
- Straightforward to code
- Reliable

CFDLab

Round Robin Communications

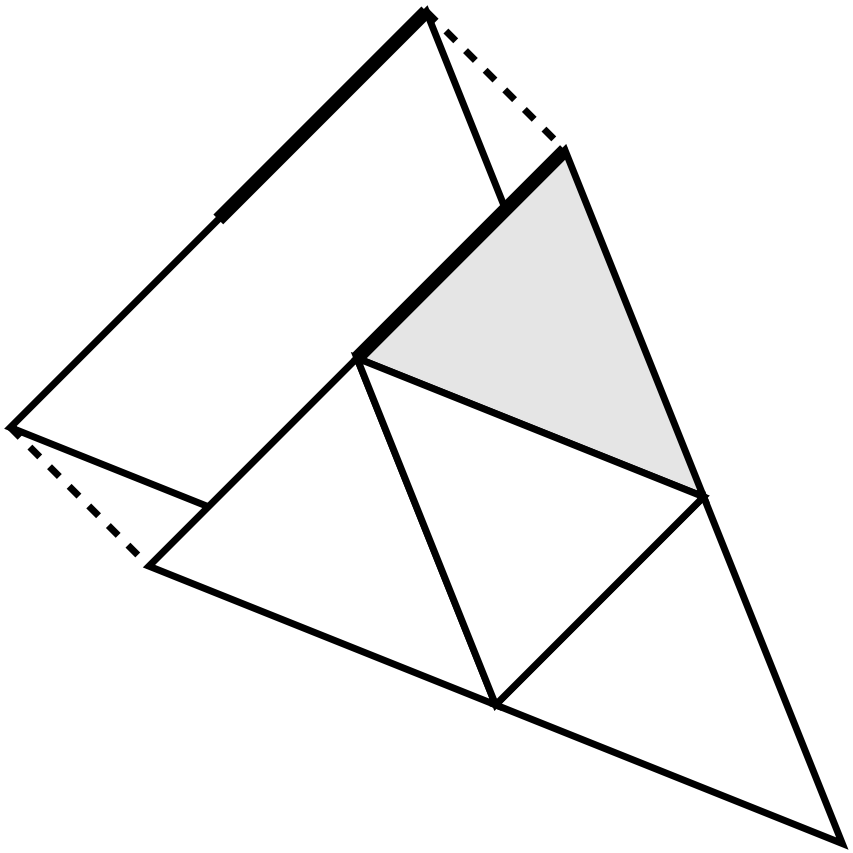
Cons

- Communications loop over non-neighboring processors
- $O(N_G)$ execution time
- Multiple, synchronous communications



CFDLab

Adaptivity and ParallelMesh



Challenges

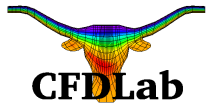
- Reindexing elements, nodes, DoFs
- Synchronization of ghost objects
- Load balancing, Repartitioning

CFDLab

Parallel Global Indexing

One-pass indexing

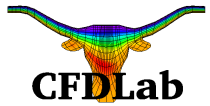
- Index processor P from $\sum_1^{P-1} N_{E_p}$
- Pass $\sum_1^{P} N_{E_p}$ to processor $P+1$
- $O(N_E)$ work
- $O(N_E)$ execution time



Parallel Global Indexing

Two-pass indexing

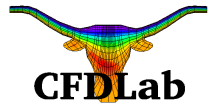
- Count processor P indices from 0
- Gather N_{E_p} on all processors
- Re-index processor P from $\sum_1^{P-1} N_{E_p}$
- Double the work
- $O(N_E/N_P)$ execution time



Parallel Synchronization

What can lose sync?

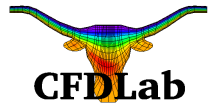
- Refinement flags
- New child elements, nodes
- New degrees of freedom
- Hanging node constraint equations
- Repartitioned elements, nodes



Parallel Synchronization

Round-Robin Complications

- Refinement flags must obey consistency rules
- New ghost nodes may have unknown processor ids
- Constraint equations may be recursive
- Hanging node constraint equations
- Repartitioned elements, nodes



Debugging

- Regression tests
- Precondition, postcondition tests
- Unit testing
- Parallel debuggers
- Low N_P test cases

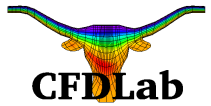
Summary

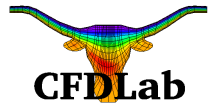
Parallelism tradeoffs

- Efficiency vs. ease of programming/debugging
- Latency vs. redundant work
- “Premature optimization” mistakes vs. bad assumptions

and guidelines

- Reuse existing code/algorithms
- Build incrementally
- Test extensively





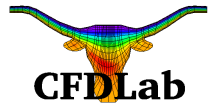
Mesh Generation

- You will need access to a complete mesh generation package to create complex meshes for use inside `libMesh`.
- General process involves importing geometry, creating a volume mesh, assigning boundary conditions, and exporting the mesh.
- Recommended mesh generation packages:
  - `gridgen/pointwise`: export mesh in `ExodusII` format.
  - `gmsh`: open-source mesh generator. `libMesh` supports `gmsh` format.
  - `Cubit`: unstructured mesh generator from Sandia National Labs.
  - Need another mesh format? `libmesh-users@lists.sourceforge.net`

CFDLab

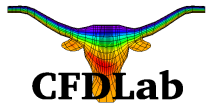
Discontinuous Galerkin Support

- `libMesh` supports a rich set of discontinous finite element bases.
- For an example using interior penalty DG, see `$LIBMESH_DIR/examples/miscellaneous/ex5`



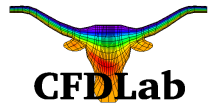
Multiphysics Support

- Tightly coupled multiphysics:
  - All variables should be placed in the same `System`.
  - You can restrict variables to subdomains: c.f. `$LIBMESH_DIR/examples/subdomains/ex1`
- Loosely coupled multiphysics:
  - On the same `Mesh`, use different `Systems`
  - On disjoint `Mesh`es, pass data via `MeshfreeInterpolation` objects.



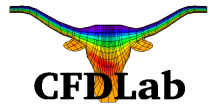
FEMSystem

- `FEMSystem` provides an alternative programming interface specifically for finite element applications.
- Allows more direct interaction between solution algorithms and finite element approximation.
- See `$LIBMESH_DIR/examples/fem_system`



Reduced Basis Modeling

- Recent effort led by David Knezevic to add certified reduced basis support to `libMesh`.
- This functionality allows fine-scale solutions to be approximated in an optimal basis with an associated error estimate.
- See `$LIBMESH_DIR/examples/reduced_basis`



- B. Kirk, J. Peterson, R. Stogner and G. Carey, “libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations”, *Engineering with Computers*, vol. 22, no. 3–4, p. 237–254, 2006.
- Public site, mailing lists, SVN tree, examples, etc.: `http://libmesh.sf.net/`

CFDLab