LibMesh: A Parallel Adaptive Finite Element Library

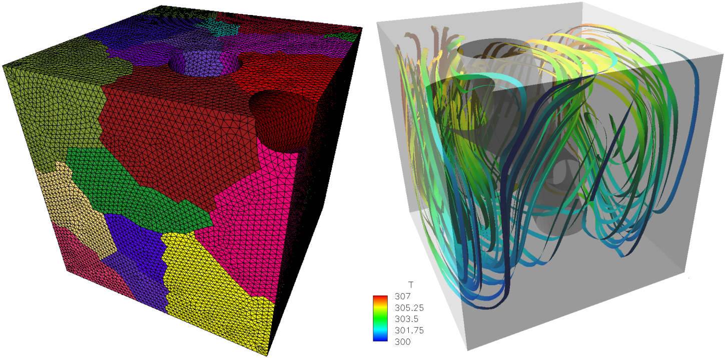
John W. Peterson

peterson@tacc.utexas.edu

Texas Advanced Computing Center
University of Texas at Austin

May 21, 2009

- An open-source (LGPL) finite element library.
- “Lowers the barrier of entry” to serial and parallel simulation of multi-scale, multi-physics applications.
- Implements adaptive mesh refinement and coarsening on unstructured, hybrid grids in 1, 2, and 3D.
- **`libmesh.sf.net`**



- Tetrahedral mesh of “pipe” geometry. Stream ribbons colored by temperature.

- Originated in the CFDLab, UT-Austin.
- Started by *B. Kirk* (now at NASA) as part of PhD research.
- *J. Peterson* (TACC), first user, early class design/organization.
- Current lead developer is *R. Stogner* (ICES Post-doc, UT-Austin).
- Common PhD advisor: *Graham F. Carey*

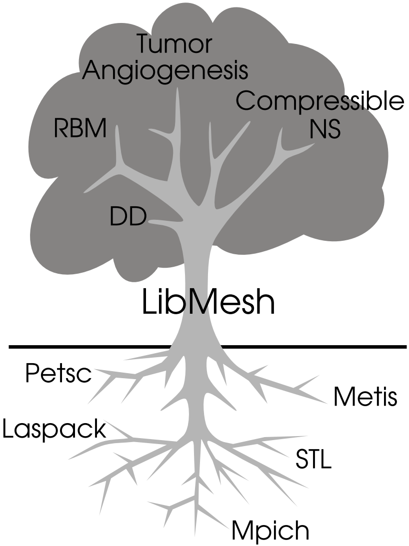
- Daniel Dreyer and Steffen Petersen, TUHH (Infinite Elements)
- Derek Gaston, INL (Mesh Redistribution)
- David Knezevic, MIT (1D support, Adaptive Mesh I/O)
- Tim Kröger, Universität Bremen (Shell Matrices, Ghosted Vectors)
- Many others: A. Coutinho, O. Certik, M. Lüthi, W. Ruijter, J. Roman, V. Mahadevan, V. Garg, V. Carey, . . .

LibMesh is not

- A physics implementation.
- A stand-alone application.

LibMesh is

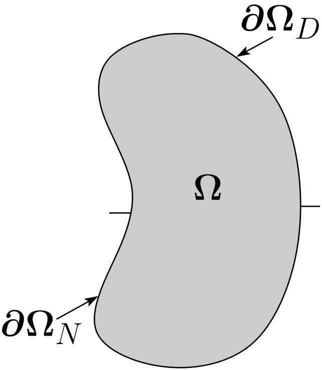
- A software library and toolkit.
- Classes and functions for writing parallel adaptive finite element applications.
- An interface to linear algebra, meshing, partitioning, etc. libraries.



- Basic libraries are LibMesh's "roots".
- Application "branches" built off the library "trunk".

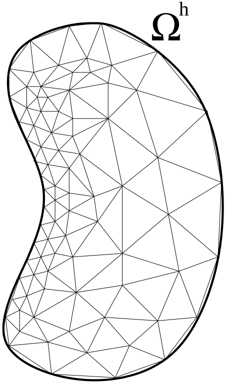
- General boundary value problems of the form:

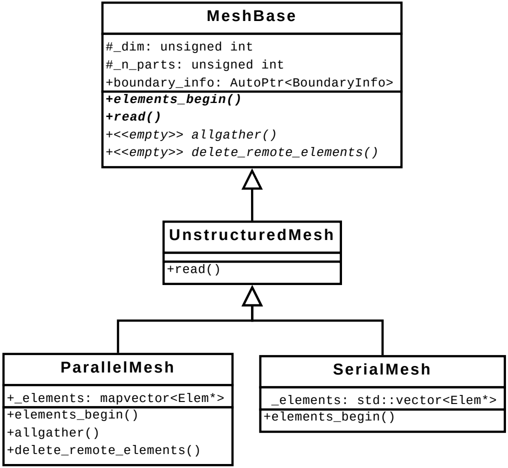
$$
\begin{aligned}
M\frac{\partial u}{\partial t} &= F(u) && \in \Omega \\
G(u) &= 0 && \in \Omega \\
u &= u_D && \in \partial\Omega_D \\
N(u) &= 0 && \in \partial\Omega_N \\
u(\boldsymbol{x},0) &= u_0(\boldsymbol{x})
\end{aligned}
$$



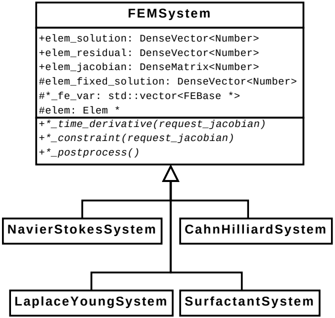
- Associated to the problem domain Ω is a LibMesh data structure called a `Mesh`
- A `Mesh` is essentially a collection of finite elements

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





- Serial `Mesh` recently extended to Parallel
- Serial functionality still present & inter-changeable



- User provides (weak form) weighted residuals

$$\left(M\frac{\partial u}{\partial t}, v_i\right) = \left(F\left(u\right), v_i\right)$$

- And/or constraints

$$\left(G\left(u\right), v_i\right) = 0$$

- As a representative example, consider the weak form arising from the Poisson equation,

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

- 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.

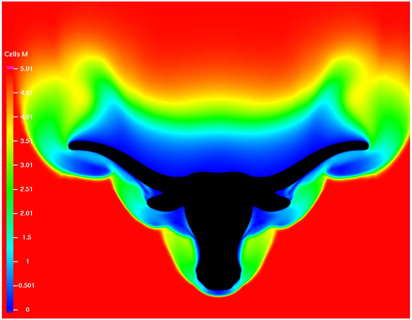
```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

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

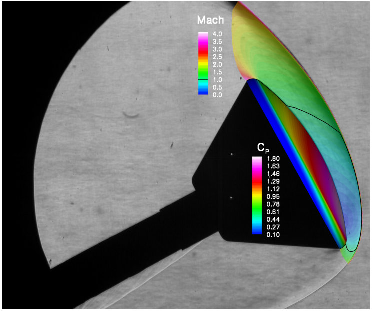
$$\boldsymbol{F}_i^e = \sum_{q=1}^{N_q} f(x(\xi_q))\phi_i(\xi_q)|J(\xi_q)|w_q$$

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

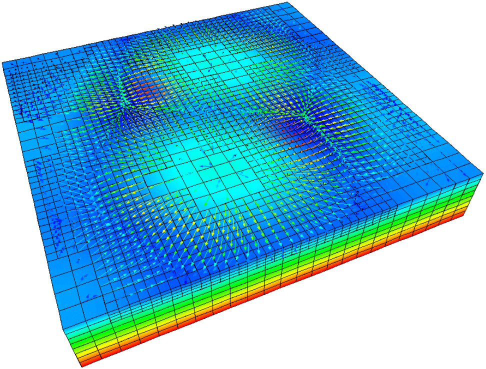
```
for (q=0; q<Nq; ++q)
  for (i=0; i<Ns; ++i) {
    Fe(i)    += JxW[q]*f(xyz[q])*phi[i][q];

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

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

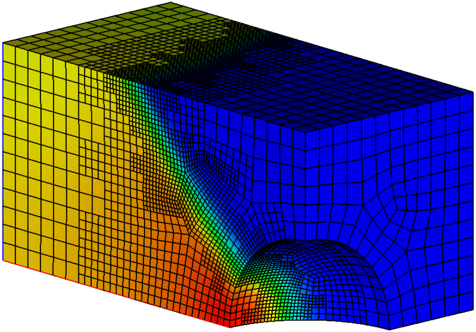




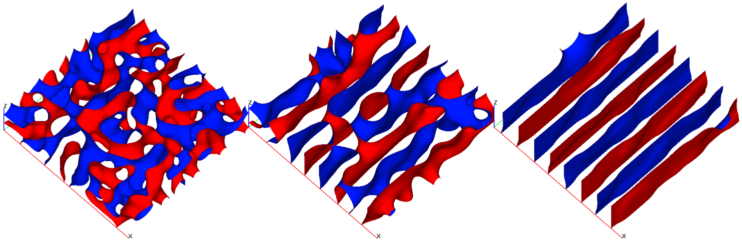
- LibMesh is being used in the development of the Orion CEV at NASA.



- Adaptive grid solution shown with temperature contours and velocity vectors.



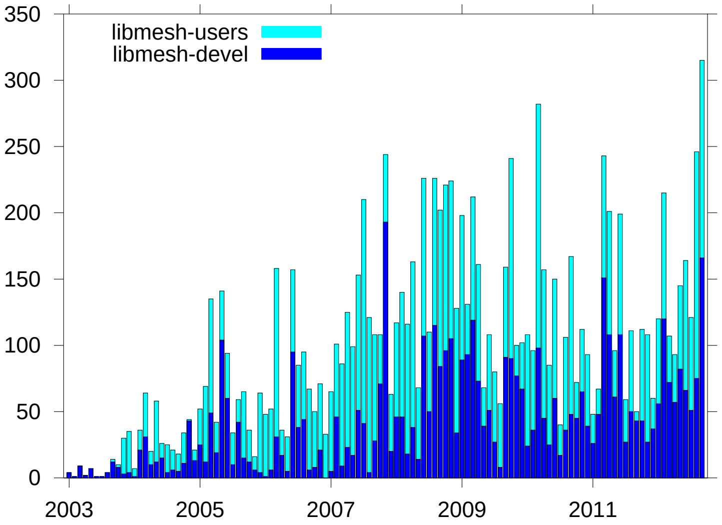
- A sharp advancing endothelial cell front approaches the tumor, eventually inducing blood vessel formation.

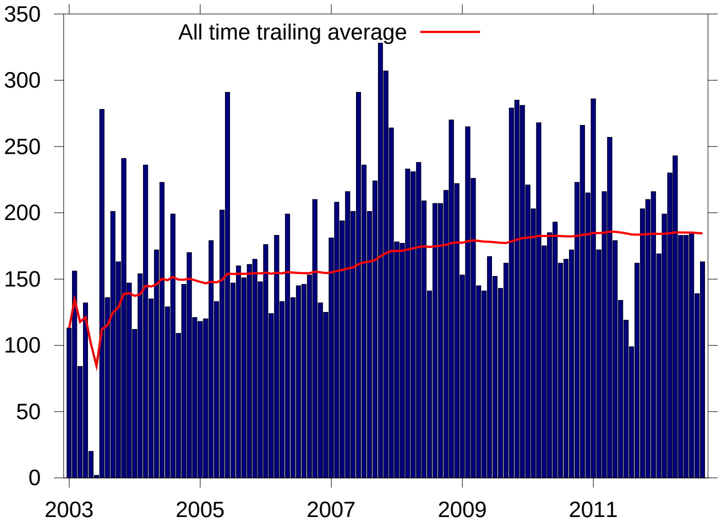


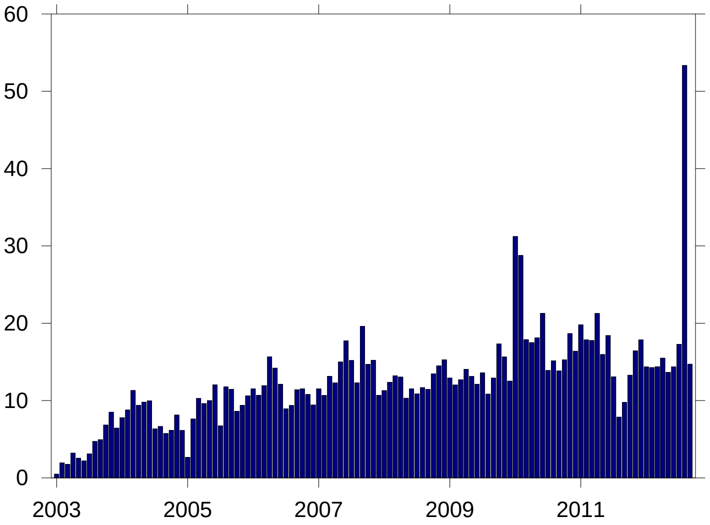
- Single-phase regions gradually coalesce
- Patterning may occur when additional stresses are present

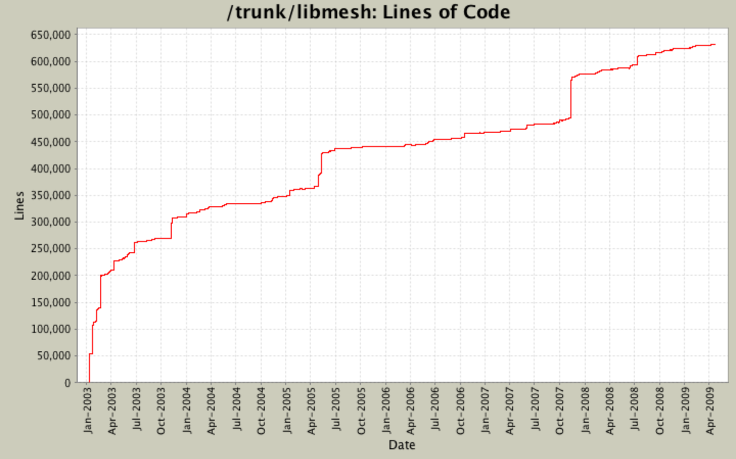
- B. Kirk, J. Peterson, R. Stogner and G. Carey, "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations," *Engineering with Computers*, vol. 22, no. 3–4, p. 237–254, 2006.

- J. W. Peterson, G. F. Carey, and B. T. Murray, "Multi-Resolution Simulation of Double-Diffusive Convection in Porous Media," *Int. J. Numer. Meth. for Heat & Fluid Flow*, to appear, Nov. 2009 issue.
- J. Biermann, O. von Estorff, S. Petersen, C. Wenterodt, "Higher order finite and infinite elements for the solution of Helmholtz problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 198, no. 13–14, p. 1171–1188, 2009.
- B. Kirk, "A Multidimensional Thermal Analysis to Assess Modeling Error in High-Speed Wind Tunnel Heat Transfer Data Reduction Schemes," *AIAA Journal of Thermophysics and Heat Transfer*, vol. 23, no. 1, p. 186–191, 2009.
- Benjamin S. Kirk, Graham F. Carey, "Development and Validation of a SUPG Finite Element Scheme for the Compressible Navier-Stokes Equations using a Modified Inviscid Flux Discretization," *Int. J. Numer. Meth. Fluids*, vol. 57, no. 3, p. 265–293, 2008.
- R. Stogner and G. Carey, "C^1 macroelements in adaptive finite element methods," *Int. J. Num. Meth. Eng.*, vol. 70, no. 9, p. 1076–1095, 2007.
- M. F. Barone, I. Kalashnikova, D. J. Segalman, H. K. Thornquist, "Stable Galerkin reduced order models for linearized compressible flow," *Journal of Computational Physics*, vol. 228, no 6, p. 1932–1946 2009.
- M. Piccinelli, L. Botti, B. Ene-Iordache, A. Remuzzi, A. Veneziani, L. Antiga, "Link between vortex structures and voronoi diagram in cerebral aneurysms," *Journal of Biomechanics*, vol. 41, Supplement 1, p. S12, 2008.
- F. Galbusera, M. Cioffi, and M. T. Raimondi, "An in silico bioreactor for simulating laboratory experiments in tissue engineering," *Biomedical Microdevices*, vol. 10, no. 4, p. 547–554, August, 2008.
- M. Anderson, J.-H. Kimn, "A numerical approach to space-time finite elements for the wave equation," *Journal of Computational Physics*, vol. 226, no. 1, p. 466–476, 2007.


libmesh-users
libmesh-devel
350
300
250
200
150
100
50
0
2003
2005
2007
2009
2011


All time trailing average
350
300
250
200
150
100
50
0
2003
2005
2007
2009
2011


60
50
40
30
20
10
0
2003
2005
2007
2009
2011



- Lines of code vs. time over the life of the library.

- Wait, this is supposed to be “Scientific Software.” Isn’t OO code ‘slow’?

- Wait, this is supposed to be “Scientific Software.” Isn’t OO code ‘slow’?

Yes!

- Wait, this is supposed to be “Scientific Software.” Isn’t OO code ‘slow’?

Yes!

- But, this is like asking if driving a car is ‘dangerous’.

- Wait, this is supposed to be “Scientific Software.” Isn’t OO code ‘slow’?

  Yes!

- But, this is like asking if driving a car is ‘dangerous’.

- It is dangerous, but it is also a very convenient and effective means of transportation.

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

  Yes!

- But, this is like asking if driving a car is 'dangerous'.
- It is dangerous, but it is also a very convenient and effective means of transportation.
- As long as everyone plays by the rules, nobody gets hurt!

- Consider a simple example using a vector to implement row-major storage.

```
long matrix_size = 10000;
std::vector<double> v(matrix_size*matrix_size);

long cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    v[i*matrix_size+j] = cnt++;
```

- Consider a simple example using a vector to implement row-major storage.

```
long matrix_size = 10000;
std::vector<double> v(matrix_size*matrix_size);

long cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    v[i*matrix_size+j] = cnt++;
```

- We can instead hide the index calculation in a user-defined Matrix type.

```
class Matrix
{
public:
  Matrix(int mm, int nn);
  double& operator()(int i, int j);
private:
  int m, n;
  std::vector<double> vals;
};
```

- The user code is now:

```
long matrix_size = 10000;
Matrix m(matrix_size,matrix_size);

long cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    m(i,j) = cnt++;
```

- Timing results (in seconds, averaged over 5 runs) for the two different versions with different optimization levels.

	None	**–O3**
std::vector	5.44	1.72
Matrix	6.10	1.70

- With a decent compiler (in this case, `g++`) there is almost no difference in performance between the two.

- Does not require much advanced optimization knowledge on the part of the user (e.g. expression templates).

- The "object" code is arguably cleaner, and provides better reuse possibilities.

- Virtual functions are another OO feature frequently cited as “slow.”
- Consider our previous Matrix class modified to allow subclassing:

```
class MatrixBase
{
public:
  MatrixBase(int mm, int nn);
  virtual ~MatrixBase() {}
  virtual double& operator()(int i, int j) = 0;
protected:
  int m, n;
  std::vector<double> vals;
};
```

- Define the MatrixRowMajor subclass to implement row-major storage:

```
class MatrixRowMajor : public MatrixBase
{
public:
  MatrixRowMajor(int mm, int nn);
  virtual double& operator()(int i, int j);
};

double& MatrixRowMajor::operator()(int i, int j)
{
  return vals[i*n + j]; // row major
}
```

- Define the MatrixColMajor subclass for column-major storage:

```
class MatrixColMajor : public MatrixBase
{
public:
  MatrixColMajor(int mm, int nn);
  virtual double& operator()(int i, int j);
};

double& MatrixColMajor::operator()(int i, int j)
{
  return vals[i + m*j]; // col major
}
```

- (Essentially) the same matrix-fill code can be re-used...

```
// Create row-major (or col) matrix...
MatrixBase& m =
  *(new MatrixRowMajor(matrix_size,matrix_size));

long cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    m(i,j) = cnt++;
```

- Average timing results (in seconds) for the original and polymorphic Matrix classes.

	None	**–O3**
Matrix	6.10	1.70
Matrix (virtual, row-major)	6.08	1.98
Matrix (virtual, col-major)	8.06	3.77

- The additional flexibility obtained by decoupling the storage layout from the algorithm cost us about 15% in the row-major case.
- Also, the "generic" algorithm (which is inherently row-major) did not perform nearly as well on the column-major layout.
- We can address both these issues by becoming virtual at a "higher level."

- Recognizing that the algorithm is not efficiently decoupled from the storage layout, we can make the *algorithm itself* virtual instead.

```
class MatrixBase
{
public:
  MatrixBase(int mm, int nn);
  virtual ~MatrixBase() {}
  virtual void fill() = 0;
protected:
  int m, n;
  std::vector<double> vals;
};
```

- Implemented for the row-major case (col-major case is analogous):

```
void MatrixRowMajor::fill()
{
  long cnt=0;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
      vals[i*n + j] = cnt++;
}
```

- And finally, called generically from user code:

```
MatrixBase* m =
  new MatrixRowMajor(matrix_size,matrix_size);
m->fill();
```

- Combined results for the original, non-virtual objects and the virtual `fill()` function.

	None	**–O3**
std::vector	5.44	1.72
Matrix	6.10	1.70
fill(), row-major	5.70	1.68
fill(), col-major	5.71	1.69

- Proper use of virtual functions (i.e. not too many) leads to more flexible code with the same performance as less flexible code.

- The `fill()` method in this example can be made more sophisticated if we also pass a "`Filler`" function object to it.

- This example was trivial: there are libraries (boost/blitz/eigen) which are much more realistic.

- The guidelines developed here for using virtual functions should apply in other situations as well.