The Finite Element Method in Scientific Computing
MATH 9830

Timo Heister
(heister@clemson.edu)

http://www.math.clemson.edu/~heister/math983-spring2014/

Goals of this course

- Learn about the software library deal.II
- understand practical aspects of finite element software
- use the library deal.II for own computations
- Build, document, and present a sophisticated software project
- solve nonlinear, time-dependent, and coupled PDEs
- use advanced tools in software development (IDEs, debuggers, ...)
- Not:
  - Teach the Finite Element Method (see 8660)
  - Learn how to program in C++

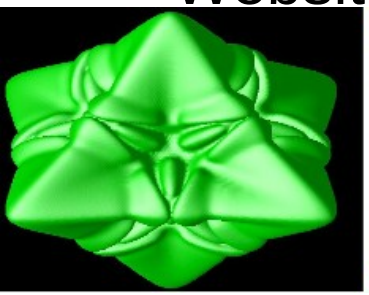
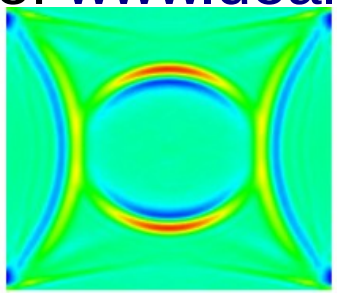
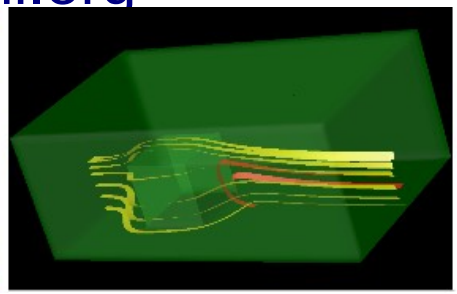
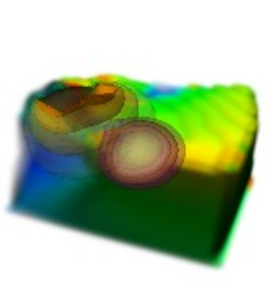
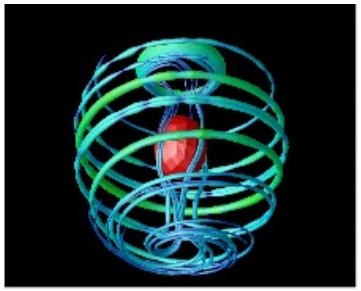
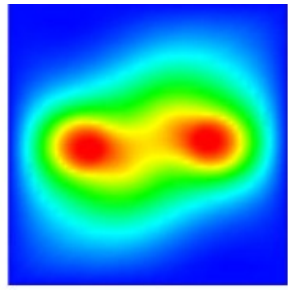
  (but I am happy to help!)

Topics

- basics of FEM, structure of FEM codes, algorithmic aspects
- modern tools for software development (IDEs, debuggers, …)
- some C++ topics (templates, …) used in large software projects
- iterative solvers and preconditioners
- coupled PDEs, block systems
- nonlinear problems
- time discretization
- parallel computations
- software engineering practices

deal.II

- “A Finite Element **D**ifferential **E**quations **A**nalysis **L**ibrary”
- Open source, c++ library
- I am one of the three maintainers
- One of the most widely used libraries:
  - ~400 papers using and citing deal.II
  - ~600 downloads/month
  - 100+ people have contributed in the past 10 years
  - ~600,000 lines of code
  - 10,000+ pages of documentation
- Website: www.dealii.org

Features

- 1d, 2d, 3d computations, adaptive mesh refinement (on quads/hexas only)
- Finite element types:
  - Continuous and DG Lagrangian elements
  - Raviart-Thomas, Nedelec, …
  - Higher order elements, hp adaptivity
  - And arbitrary combinations

Features, part II

- Linear Algebra
  - Own sparse and dense library
  - Interfaces to PETSc, Trilinos, UMFPACK, BLAS, ..
- Parallelization
  - Multi-threading on multi-core machines
  - MPI: 16,000+ processors
- Output in many visualization file formats

Development of deal.II

- Professional-level development style
- Development in the open, open repository
- Mailing lists for users and developers
- Test suite with 6,000+ tests after every change
- Platform support:
  - Linux/Unix
  - Mac
  - Work in progress: Windows

For you

- Join the mailing lists
  - Ask questions
  - Just read and learn
- Become a contributor
  - Smallest changes are welcome! (find a typo? Documentation of a function lacking? Implement a small feature?)
- Cite deal.II if you use it

Homework 1

- Due on Friday:
  - Create google document, share with timo.heister@gmail.com and start taking notes!
  - Watch lecture 1
- Setup and bring a laptop running Linux (preferred), or Mac OS
  - Dual booting Windows and Ubuntu possible
  - I am happy to help

Installation

- How to install from source code, configure, compile, test, run “step-1”
- Ubuntu (or any other linux) or Mac OSX
- Steps:
  - Detect compilers/dependencies/etc. (cmake)
  - Compile & install deal.II (make)

Prerequisites on Linux

- Compiler: GNU g++
- Recommended:

```
$ sudo apt-get install subversion openmpi1.6-bin
openmpi1.6-common g++ gfortran libopenblas-dev
liblapack-dev zlib1g-dev git emacs gnuplot
```

- manually: cmake (in a minute)
- Later: eclipse, paraview
- Optional manually: visit, p4est, PETSc, Trilinos, hdf5

On Mac OS

- See https://code.google.com/p/dealii/wiki/MacOSX
- Install xcode
- Install command line tools (under under preferences->Downloads->Components, or xcode-select –install depending on version)
- If OSX 10.9 follow instructions from wiki, else from source using:
- Cmake: download Mac OSX 64/32-bit Universal .dmg file from http://www.cmake.org/cmake/resources/software.html , click to install, hit "install links into /local/bin"
- Later manually: eclipse, paraview
- Optionally: ...

cmake

- Ubuntu 12.04 has a version that is too old
- If newer ubuntu do:

```
$ sudo apt-get install cmake
```

  … and you are done
- Otherwise: install cmake from source or download the 32bit binary

cmake from binary

- Do:

```
export CMAKEVER=2.8.12.1

wget http://www.cmake.org/files/v2.8/cmake-$CMAKEVER-Linux-i386.sh

chmod u+x cmake-$CMAKEVER-Linux-i386.sh

./cmake-$CMAKEVER-Linux-i386.sh
```

- Answer “q”, yes and yes
- Add the bin directory to your path (.bashrc)
- You might need

```
sudo apt-get install ia32-libs
```

Cmake from source

```
wget ¬ http://www.cmake.org/files/v2.8/cmake-2.8.12.1.tar.gz

tar xf cmake-2.8.11.1.tar.gz

./configure

make install
```

Install deal.II

- http://www.dealii.org/8.1.0/readme.html
- Extract:

  ```
  tar xf deal.II-8.1.0.tar.gz
  ```

- Build directory:

  ```
  cd deal.II; mkdir build; cd build
  ```

- Configuration:

  ```
  cmake -D CMAKE_INSTALL_PREFIX=/?/? ..
  ```

  (where /?/? is your installation directory)

- Compile (5-60 minutes):

  ```
  make -j X install
  ```

  (where X is the number of cores you have)

- Test:

  `make test` (in build directory)

- Test part two:

  ```
  cd examples/step-1

  cmake -D DEAL_II_DIR=/?/? .

  make run
  ```

- Recommended layout:

  ```
  deal.II/

      build       < build files

      installed   < your inst. dir

      examples    < all examples!

      include

      source

      ...
  ```

How to create an eclipse project

- Run this once in your project:

  cmake -G "Eclipse CDT4 - Unix Makefiles" .
- Now create a new project in eclipse (“file->import->existing project” and select your folder for the project above)

Templates in C++

- “blueprints” to generate functions and/or classes
- Template arguments are either numbers or types
- No performance penalty!
- Very powerful feature of C++: difficult syntax, ugly error messages, slow compilation
- More info:
  http://www.cplusplus.com/doc/tutorial/templates/

  http://www.math.tamu.edu/~bangerth/videos.676.12.html
-

Why used in deal.II?

- Write your program once and run in 1d, 2d, 3d:

```
DoFHandler<dim>::active_cell_iterator

    cell = dof_handler.begin_active(), endc =
dof_handler.end();

for (; cell!=endc; ++cell)

 { ...

      cell_matrix(i,j) += (fe_values.shape_grad (i, q_point)
*

                                    fe_values.shape_grad (j,
q_point) *

                                    fe_values.JxW (q_point));
```

- Also: large parts of the library independent of dimension

Function Templates

- Blueprint for a function
- One type called “number”
- You can use “typename” or “class”
- Sometimes you need to state which function you want to call:

```
template <typename number>
number square (const number x)
{ return x*x; };

int x = 3;
int y = square(x);
```

```
template <typename T>
void yell ()
 { T test; test.shout("HI!"); };

// cat is a class that has shout()
yell<cat>();
```

Value Templates

- Template arguments can also be values (like int) instead of types:

```
template <int dim>
void make_grid (Triangulation<dim> &triangulation)
{ …}

Triangulation<2> tria;
make_grid(tria);
```

- Of course this would have worked here too:

```
template <typename T>
void make_grid (T &triangulation)
{ ...
```

Class templates

- Whole classes from a blueprint
- Same idea:

```
template <int dim>
class Point
{
  double elements[dim];
  // ...
}

Point<2> a_point;
Point<5> different_point;
```

```
namespace std
{
  template <typename number>
  class vector;
}

std::vector<int> list_of_ints;
std::vector<cat> cats;
```

Example

```
template <unsigned int N>
double norm (const Point<N> &p)
{
 double tmp = 0;
 for (unsigned int i=0; i<N; ++i)
   tmp += square(v.elements[i]);
 return sqrt(tmp);
};
```

- Value of N known at compile time
- Compiler can optimize (unroll loop)
- Fixed size arrays faster than dynamic

  (dealii::Point<dim> vs dealii::Vector<double>)

Examples in deal.II

- Step-4:

```
template <int dim>
void make_grid (Triangulation<dim> &triangulation) {...}
```

- So that we can use Vector<double> and Vector<float>:

```
template<typename number>
class Vector< number > { ...};
```

- Default values (embed dim-dimensional object in spacedim):

```
template<int dim, int spacedim=dim>
class Triangulation< dim, spacedim > { ...};
```

- Already familiar:

```
template<int dim, int spacedim>
void GridGenerator::hyper_cube  (Triangulation< dim, spacedim > & tria, const
double left, const double right) {...}
```

Explicit Specialization

- different blueprint for a specific type T or value

```
// store some information
// about a Triangulation:

template <int dim>
struct NumberCache
{};

template <>
struct NumberCache<1>
{
  unsigned int n_levels;
  unsigned int n_lines;
}
```

```
template <>
struct NumberCache<2>
{
  unsigned int n_levels;
  unsigned int n_lines;
  unsigned int n_quads;
}

// more clever:
template <>
struct NumberCache<2>:
public NumberCache<1>
{
      unsigned int n_quads;
}
```

step-4

- Dimension independent Laplace problem
- Triangulation<2>, DoFHandler<2>, …

  replaced by

  Triangulation<dim>, DoFHandler<dim>, …
- Template class:

```
template <int dim>
class Step4 {};
```

Adaptive Mesh Refinement

- Typical loop:
  - Solve
  - Estimate
  - Mark
  - Refine/coarsen
- Estimate is problem dependent:
  - Approximate gradient jumps: `KellyErrorEstimator` class
  - Approximate local norm of gradient: `DerivativeApproximation` class
  - Or something else
- Mark:

  `GridRefinement::refine_and_coarsen_fixed_number(...)` or

  `GridRefinement::refine_and_coarsen_fixed_fraction(...)`
- Refine/coarsen:
  - `triangulation.execute_coarsening_and_refinement ()`
  - Transferring the solution: `SolutionTransfer` class (maybe discussed later)

Constraints

- Used for hanging nodes (and other things!)
- Have the form:

$$x_i = \sum_j \alpha_{ij} x_j + c_j$$

- Represented by class ConstraintMatrix
- Created using `DoFTools::make_hanging_node_constraints()`
- Will also use for boundary values from now on:

```
VectorTools::interpolate_boundary_values(..., constraints);
```

- Need different SparsityPattern (see step-6):

```
DoFTools::make_sparsity_pattern (..., constraints, ...)
```

Constraints II

- Old approach (explained in video):
  - Assemble global matrix
  - Then eliminate rows/columns: `ConstraintMatrix::condense(...)`

    (similar to `MatrixTools::apply_boundary_values()` in step-3)
  - Solve and then set all constraint values correctly: `ConstraintMatrix::distribute(...)`
- New approach (step-6):
  - Assemble local matrix as normal
  - Eliminate while transferring to global matrix:

```
constraints.distribute_local_to_global (cell_matrix, cell_rhs,
                                        local_dof_indices,
                                        system_matrix, system_rhs);
```

  - Solve and then set all constraint values correctly: `ConstraintMatrix::distribute(...)`

Vector Values Problems

- (video 19&20)
- FESystem: list of FEs (can be nested!)
- Will give one FE with N shape functions
- Use FEValuesExtractors to do
  `fe_values[velocities].divergence (i, q), ...`
- Ordering of DoFs in system matrix is independent
- See module “handling vector valued problems”
- Non-primitive elements (see fe.is_primitive()):

  shape functions have more than one non-zero component, example: RT

Computing Errors

- Important for verification!
- See step-7 for an example
- Set up problem with analytical solution and implement it as a Function<dim>
- Quantities or interest:

$$e = u - u_h$$

$$\|e\|_0 = \|e\|_{L_2} = \left(\sum_K \|e\|_{0,K}^2 \right)^{1/2} \qquad \|e\|_{0,K} = \left(\int_K |e|^2 \right)^{1/2}$$

$$|e|_1 = |e|_{H^1} = \|\nabla e\|_0 = \left(\sum_K \|\nabla e\|_{0,K}^2 \right)^{1/2}$$

$$\|e\|_1 = \|e\|_{H^1} = \left(|e|_1^2 + \|e\|_0^2 \right)^{1/2} = \left(\sum_K \|e\|_{1,K}^2 \right)^{1/2}$$

- Break it down as one operation per cell and the “summation” (local and global error)
- Need quadrature to compute integrals

Computing Errors

- Example:

```
Vector<float> difference_per_cell (triangulation.n_active_cells());
VectorTools::integrate_difference (dof_handler,
                                   solution, // solution vector
                                   Solution<dim>(), // reference solution
                                   difference_per_cell,
                                   QGauss<dim>(3), // quadrature
                                   VectorTools::L2_norm); // local norm
const double L2_error = difference_per_cell.l2_norm(); // global norm
```

- Local norms:

  `mean, L1_norm, L2_norm,Linfty_norm, H1_seminorm, H1_norm, ...`

- Global norms are vector norms: `l1_norm(), l2_norm(), linfty_norm(), ...`

ParameterHandler

- Control program at runtime without recompilation
- You can put in:

  ints (e.g. number of refinements), doubles (e.g. coefficients, time step size), strings (e.g. choice for algorithm/mesh/problem/etc.), functions (e.g. right-hand side, reference solution)
- Stuff can be grouped in sections
- See class-repository: prm/

ParameterHandler

```
# order of the finite element to use.
set fe order   = 1

# Refinement method. Choice between 'global' and 'adaptive'.
set refinement = global

subsection equation
  # expression for the reference solution and boundary values. Function of x,y (and z)
  set reference = sin(pi*x)*cos(pi*y)

  # expression for the gradient of the reference solution. Function of x,y (and z)
  set gradient  = pi*cos(pi*x)*cos(pi*y); -pi*sin(pi*x)*sin(pi*y)

  # expression for the right-hand side. Function of x,y (and z)
  set rhs       = 2*pi*pi*sin(pi*x)*cos(pi*y) + sin(pi*x)*cos(pi*y)
end
```