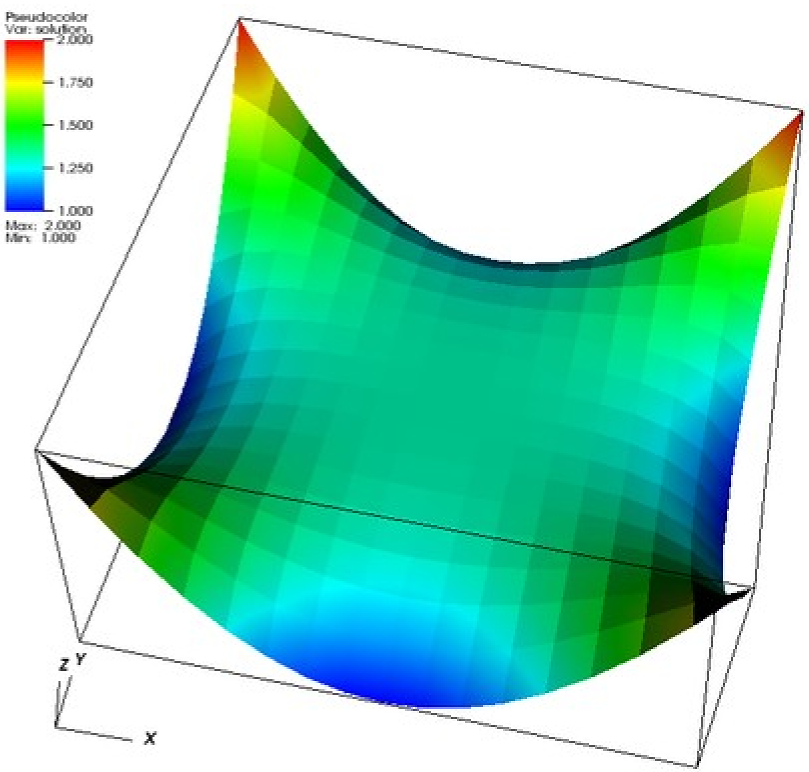
deal.II Users and Developers Training

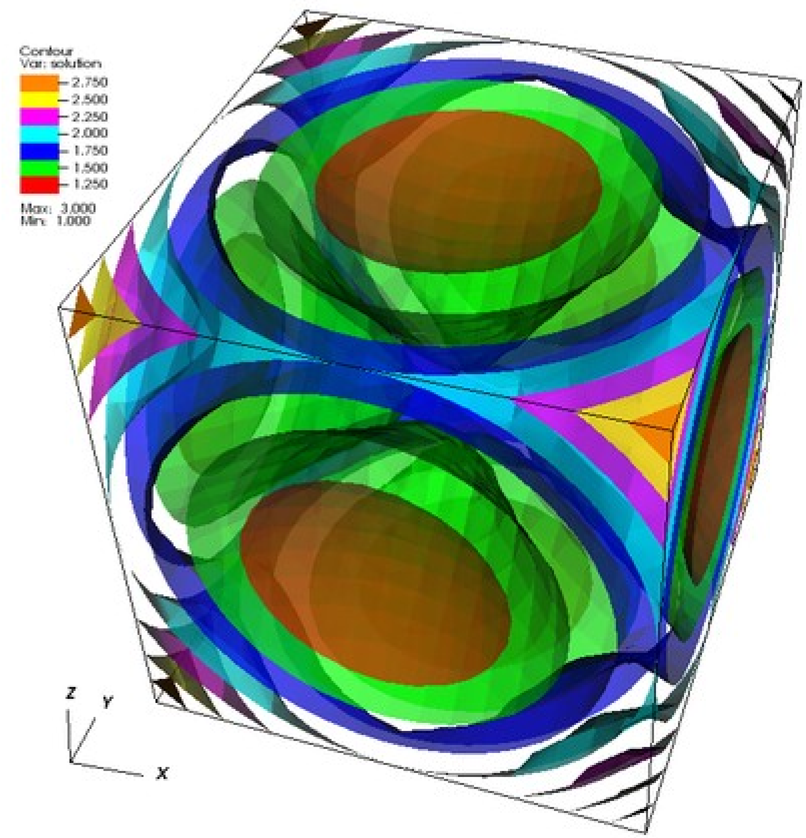
March 21 – 24 2016

Timo Heister (heister@clemson.edu)
Luca Heltai (luca.heltai@sissa.it)

Towards Lab 4 (step-4)

- Goals:
  - Dimension independent programming
  - Need: C++ templates





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
- Demos in /scratch/smr2909/lab-04/

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));
```

- cell->face () is a quad in 3d but a line in 2d
- Also: large parts of the library independent of dimension:
  - hyper_cube (square vs box), etc.

Class Templates for Functions

- <u>Blueprint</u> for a function
- One type called “number”
- You can use “typename” or “class”

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

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

- Sometimes you need to state which function you want to call:

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

- Of course this would have worked here too:

```
template <typename T>
void make_grid (T &triangulation)
{ …// now we can not access "dim" though
```

Class templates

- Whole classes instead of functions built 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, never stored!
- 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 > { number [] elements; ...};
```

- 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;
}
```

Lab 4 (step-4)

- Dimension independent Laplace problem
- Triangulation<2>, DoFHandler<2>, …
  replaced by
  Triangulation<dim>, DoFHandler<dim>, …
- Template class:

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

Lab 5

- Modified step-4 to check correctness
- Using the method of manufactured solutions
- Computing L2 and H1 errors and check orders

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(), ...`

Lab 6

- Higher order mappings, see step-10/step-11
- Start with lab-6. Find a solution so that higher order mapping gives correct convergence order!