

A brief overview of deal.II

Wolfgang Bangerth, TICAM

What is deal.II?

What deal.II is and has:

- a general purpose finite element library
- provides easy-to-use ways to handle adaptive grids and to implement error estimators and other advanced things
- provides a number of different finite elements:
 - Q1 -- Q4
 - DQ0 -- DQ4
 - DP0 -- ...
 - Nedelec edge elements
 - composed elements
- 1d, 2d, 3d all with the same interface: write and test code only once, then compile for some space dimension
- large number of tool classes and functions
- 14 example programs, vast documentation

What deal.II is not:

- A blackbox solver for one/several particular problems!

What is deal.II?

What deal.II is presently used for:

- DGFEM for elliptic problems
- DGFEM for Stokes/Navier-Stokes
- DGFEM for Euler/hyperbolic transport
- second order (time dependent) wave equations
- inverse and optimization problems
- elastoplasticity
- porous media flow
- large deformation metal forming simulation
- biomechanical modeling
- the work here at the CFD lab

Basic ideas of deal.II

deal.II is built around a number of paradigms. The most important one is

dimension independent programming

This means:

- write your code as you think mathematically:
 - in cells/faces etc, not in quads/hexes
 - in points, not in 2-vectors, or 3-vectors
- the library provides you with classes/functions that act on objects that represent cells/faces, finite elements, and triangulations in d space dimensions, etc
- use template magic and the compiler to then compile for *one particular* space dimension
- since at compile time space dimension is known, optimizations on this can be performed by compiler (no virtual function calls, no run-time checks for dim., etc)

Dimension independent programming

Matrix assembly: $A_{ij} = \sum_K (\nabla \varphi_i, \nabla \varphi_j)_K$

```
for (cell=begin; cell!=end; ++cell)
  apply  $d$ -dimensional quadrature formula
  to  $\nabla \varphi_{i,j}$  on cell  $K$ 
```

Simple error indicator: $\eta_K^2 = \frac{h}{24} \left\| [\mathbf{n} \cdot \nabla u_h] \right\|_{\partial K}^2$

```
for (all cells)
  for (all faces of this cell)
    apply (d-1)-dimensional quadrature formula
    to jump term on this face
```

Dimension independent programming

How this looks like for the second example:

```
template <int dim>
void ErrorIndicator<dim>::compute_indicators ()
{
    QGauss3<dim-1> quadrature_formula;
    TriaIteratorTraits<dim>::cell_iterator cell;
    for (cell=tria.begin(); cell!=tria.end(); ++cell)
        for (int f=0;
             f<GeometryInfo<dim>::faces_per_cell;
             ++f)
            integrate_on_face (cell->face(f),
                               quadrature_formula);
};
```

Value templates in C++

Example:

```
template <unsigned int N>
class Vector {
    double elements[N];
};
```

- Class parameterized on number of elements

Use:

```
Vector<3> vector3;
```

- Generates class with three elements
- Number and position of data elements known at compile time; optimizations based on this information possible

Value template functions

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

- Actual value of N known at compile time
- Compiler will thus unroll loop
- Value templates faster than vectors of dynamic length

Points and structural objects

Point in dim space dimensions:

```
template <int dim>
class Point {
    double components[dim];
};
```

A structdim dimensional object in spacedim dimensional space:

```
template <int structdim, int spacedim>
class TriaObject {
    Point<spacedim> vertices[1<<structdim];
    ...
    Point<spacedim> vertex(unsigned int v);
};
```

Points and structural objects

```
template <int spacedim>
class Cell
    : public TriaObject<spacedim,spacedim>
{
    Cell<spacedim> &
        neighbor (unsigned int neighbor_no);

    TriaObject<spacedim-1,spacedim> &
        face (unsigned int face_no);

    ...
};
```

Traits

```
template <>
class TriaIteratorTraits<1> {
    typedef TriaIterator<Cell<1>> > cell_iterator;
    typedef void * face_iterator;
};
```

```
template <>
class TriaIteratorTraits<2> {
    typedef TriaIterator<Cell<2>> > cell_iterator;
    typedef TriaIterator<TriaObject<1,2>> > face_iterator;
};
```

Statistics I

Lines of code in deal.II (as of July 2002):

Dimension independent code (mostly algorithms)	113,000
Dimension dependent code (geometric interpretation, grid input/output)	23,000
Machine generate dimension dependent code (mostly finite elements)	7,000
Other code (linear algebra, tool classes)	37,000
Sum	180,000

Statistics II

Lines of code of an application
(adaptive wave equation solver):

Dimension independent code	13,700
Dimension dependent code (coarse grid generation + small pieces)	400
Other code (support classes)	1,400
Sum	15,500

www.dealii.org

Visit the deal.II library:

<http://www.dealii.org>

