Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn

universitätbonn

Practical Lab
Variational Methods and Inverse Problems in Imaging
Summer term 2014
Prof. Dr. M. Rumpf – A. Effland, B. Geihe, S. Simon, S. Tölkes

Introduction to the Quocmesh library

Preface

The Quocmesh library is a software collection jointly developed by the group "Modeling and Numerical Simulation", headed by Prof. Dr. Martin Rumpf, at Bonn University. The main application is Finite Element computations and related numerical algorithms. It is written in C++ (using partly features of the 2011 revision of the ISO C standard) and follows the object oriented programming paradigm.

On the most abstract level the Quocmesh library defines objects and operators that can be applied to objects. The principle of strong typing assures that objects fit together while intensive use of template programming retains flexibility.

A basic example: matrix-vector operations

The most fundamental data structure is the vector representing a mathematical vector of rational numbers. It is declared in the following way:

```
aol::Vector<double> vec ( 42 );
```

The argument specifies the length of the vector. Notice that the vector has a template parameter specifying the integral data type used to represent floating point numbers. Usually one would use **double** everywhere. Read and write access to the elements of the vector is provided by the usual array syntax `vec[i]`.

A matrix, which is an operator on vectors, can be declared like this:

```
aol::FullMatrix<double> mat ( 23, 42 );
```

This will construct a dense matrix with 23×42 entries. There are numerous other matrix types tailored for specific applications. Especially for Finite Element computations sparse matrices will be used. Notice that again the floating point data type has to be specified. As long as we used the same type for the vector and the matrix we can now apply the matrix to the vector.

```
mat.apply ( vec, dest );
```

This will perform a matrix vector multiplication of mat and vec and write the result to dest, which is just another vector. Checking if the dimensions of the matrix and the two vectors fit together might be done at runtime if implemented.

If you are interested in more elaborated examples have a look at `examples/vectorMatrixOps` (go to "Table of Projects" in the documentation provided).

Finite Element Operators

As an example we consider the following equation: For given $\alpha \in \mathbb{R}$ find $u : \bar{\Omega} \to \mathbb{R}$ such that

$$\begin{aligned} u - \alpha\Delta u &= u_0 \quad \text{in } \Omega\,, \\ \partial_\nu u &= 0 \quad \text{on } \partial\Omega\,. \end{aligned}$$

Discretization The weak formulation reads

$$\int_\Omega (u - \alpha\Delta u)\phi = \int_\Omega u_0\,\phi \qquad \text{for all } \phi \in H^{1,2}(\Omega)\,.$$

By partial integration we arrive at

$$\int_\Omega u\,\phi + \alpha \int_\Omega \nabla u \cdot \nabla\phi = \int_{\partial\Omega} \underbrace{\nabla u \cdot \nu}_{=0}\,\phi + \int_\Omega u_0\,\phi\,.$$

Let $U_0(\phi) := \int u_0\,\phi$. We now look for an approximation $U \in \mathcal{V}_h$ of u where $\mathcal{V}_h = \operatorname{span}\{\phi_1, \ldots, \phi_N\}$ is a finite dimensional space:

$$U = \sum_j \bar{U}_j\,\phi_j\,.$$

Inserting the approximation for u yields

$$\int_\Omega \sum \bar{U}_j\,\phi_j\,\phi_i + \alpha \int_\Omega \sum \bar{U}_j\,\nabla\phi_j \cdot \nabla\phi_i = U_0(\phi_i) \qquad \text{for all } i\,.$$

We now define the mass and the stiffness matrix by

$$M_{ij} := \int \phi_i\,\phi_j, \qquad L_{ij} := \int \nabla\phi_i\,\nabla\phi_j\,.$$

Finally we obtain a linear system of equations:

$$(M + \alpha L)\bar{U} = \bar{U}_0\,.$$

Matrix assembly The assembly of the Finite Element matrices will occur element based, i.e. we have to iterate over all elements of the computational grid and on each element consider every pairing of Finite Element basis functions whose support intersect with the current element. To compute the integral a numerical quadrature is required.

The generic assembly loop is implemented in an interface class FELinOpInterface:

```
for each element T
  prepareLocalMatrix( T, localMatrix )
  for i,j from 1 to numberOfLocalBasisFunctions
    globalIndex_i,j = mapLocalToGlobalIndex(i,j)
  endfor
  globalMatrix( globalIndex_i,j ) = localMatrix_i,j
endfor
```

The concrete implementation of prepareLocalMatrix depends on the specific Finite Element operator that is considered. It is therefore implemented in a class derived from FELinOpInterface and looks similar to the following generic version:

```
for each quadrature point q
  for i,j from 1 to numberOfLocalBasisFunctions
    basisfct_i = evaluateBasisFunction( i, q )
    basisfct_j = evaluateBasisFunction( j, q )
    localMatrix_i,j += basisfct_i * basisfct_j * quadratureWeight( q )
  end(for)
endfor
```

Similarly gradients may enter the integration or additional coefficients considered.

Configurator classes All information that is required to perform the above assembly is centralized in so called configurator classes. They provide among other things:

- the dimension of the problem
- the integral data type used to represent floating point numbers
- the quadrature rule
- the type of Finite Element functions used
- the type of computational grid

Here is an example for bilinear Finite Element functions on a two dimensional uniform quadrilateral grid on level 5 (i.e. it has $2^5 \times 2^5$ elements) using a Gauss quadrature rule of order 3. To keep the code readable we suggest to use **typedef** expressions.

```
const qc::Dimension DIM = qc::QC_2D;
typedef double RealType;
typedef aol::GaussQuadrature<RealType,DIM,3> QuadType;
typedef qc::QuocConfiguratorTraitMultiLin<RealType,DIM,QuadType> ConfType;
typedef typename ConfType::InitType GridType;

GridType grid ( 5 );
ConfType conf ( grid );
```