Exercise Sheet No.3 Deadline: 12. November 2013

EXERCISE 3 STATIONARY GROUNDWATER FLOW PROBLEM

The steady-state groundwater flow equation presented in the lecture is an elliptic partial equation given by the following PDE:

$$-\nabla \cdot \{\mathbf{K}(\vec{x}) \cdot (\nabla p_W - \rho_W g \vec{e}_z)\} + r_W(\vec{x}) = 0$$

Here $\mathbf{K}$ is the spatially variable conductivity, p_W the pressure distribution in the groundwater (solution of the PDE), ρ_W the density of water, g the acceleration due to gravity and r_W a source term for infiltration or drainage of water. In this exercise, you will have to implement the member function `assemble()` of the `class GroundwaterAssembler` which you can find in the file `gw_assembler.hh`. It should assemble the system-matrix and right hand side of a discretised version of the PDE. As a discretisation scheme, we choose the cell-centered finite volume method based on the grid tested in exercise 2.
The `class GroundwaterAssembler` provides the following public interface:

- **The constructor:**

```
GroundwaterAssembler( const Grid& grid,
                      const SpatialParameters& conductivity,
                      const Sources& sources,
                      const BoundaryConditions& bc )
```

  The constructor takes a reference to objects representing the grid (from exercise 2), the conductivity field, the boundary conditions and sources (provided below).

- **The method for assembling:**
  `void assemble( Matrix& A, Vector& b )`
  This method should assemble the system-matrix A and the right hand side vector b for uninitialized A and b. Use the vector and matrix implementations from exercise 1.

Furthermore, the `class GroundwaterAssembler` exports the type `GroundwaterAssembler::Matrix` in the following way

```
#define MATRIXTYPE DenseMatrix
class GroundwaterAssembler{
public:
 typedef MATRIXTYPE Matrix;
 ...
}
```

to make it possible for you to switch easily between different types of matrices later.

The boundary conditions are provided by the `class BoundaryConditions` in the file `gw_bc.hh`. It has the following public interface:

- **The constructor:**
  `BoundaryConditions( const Grid& grid, const std::string filename)`
  The constructor takes a constant reference to the grid and the name of a file which provides information about the boundary conditions. Both Dirichlet and Neumann boundary conditions should be available.

- **Type of condition:**
  `bool isDirichlet( const Vector& position, double* value=0 )`
  This method returns true if the boundary condition at the face barycenter is a Dirichlet condition. The corresponding Dirichlet value is stored in *value if it was set.

- **Retrieve Neumann condition:**
  `double getNeumann(const Vector& position)`
  Returns the face integrated normal flow out of the domain for the face with barycenter `position`.

- **Retrieve Dirichlet condition:**
  `double getDirichlet(const Vector& position)`
  Returns the Dirichlet (pressure) value at the face with barycenter `position`.

Notice that the coordinates given by `position` are floating point numbers and direct comparison with the == operator may not necessarily yield the desired results. To check whether a face is within the plane $x = 0$, implement a check like

```
const double epsilon = 1e-12;
if( position[0] < epsilon )
{...}
```

The class `BoundaryConditions` makes use of the `libconfig` library which you have come to know from exercise 2. Here is an example:

```
boundary_conditions:
{
# Define conditions depending on spatial range
Spatial = (
  {   type = "Neumann";
    xrange = "0..1";
    yrange = "1";
    zrange = "0..1";
     value = "-0.1";
  },
  {   type = "Dirichlet";
    xrange = "0..1";
    yrange = "0";
    zrange = "0..1";
     value = "0";
  }
 );
# Default conditions
Default = (
  {  type = "Neumann";
    value = "0";
  }
 );
};
```

A dummy implementation of the `class Sources` is also provided in the file `gw_bc.hh`. It implements a single virtual method

```
double operator()( const double& position )
```

In this exercise we will not use source terms. Hence, the method always returns zero. In future applications, we will inherit from this class.

Test your implementation on the following example.
Therefore, solve the linear system with the direct Gauss solver of the `DenseMatrix` class. We assume the domain to span $100m \times 100m$. For the picture `test05.tif`,

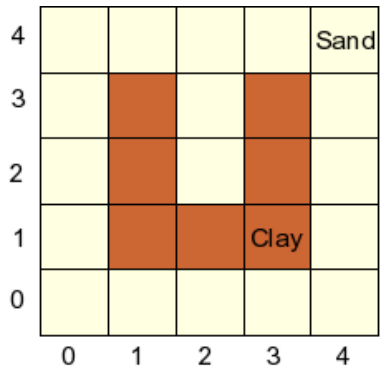

Figure 1: An artificial example of a heterogeneous ground

this corresponds to a grid resolution of $h = 20m$. Use the conductivity values from `gw_parameters.cfg`:

$$K_{\text{sand}} = 2 \cdot 10^{-5} \frac{m}{s}, \qquad K_{\text{clay}} = 2 \cdot 10^{-7} \frac{m}{s}$$

No-flux Neumann boundary conditions

$$\int_{\partial\Omega} \vec{n} \cdot \vec{J}_w ds = 0$$

are applied on the upper and lower boundary and Dirichlet conditions

$$p_{\text{left}} = 10m, \qquad p_{\text{right}} = 0m$$

on the left and right boundary.
Write the `assemble()` method to solve the problem for the resolutions

$$h_0 = 20m, \qquad h_1 = 10m, \qquad h_2 = 5m$$

and check your results using the greyscale output of the `CImg` library introduced in the exercise 2. An example for these calls is given in `gw_problem.cc`. *7 Points*