Geometric Modeling (IN4255)
TU Delft, EEMCS, Department of Intelligent Systems 2014-2015 Q. 4
K. Hildebrandt Due: 09.06.2015

Practical Assignment 2

(Shape Deformation)

Task 1 (Gradients of Linear Polynomial).

Implement a method that computes the sparse matrix G, which maps a continuous linear polynomial over all triangles of a mesh to its gradient vectors. As a first step, compute the 3×3 matrix that maps a linear polynomial over a triangle to its gradient vector. Then use this method to compute the matrix G for a triangle mesh.
Hint: On blackboard you can find meshes with a function and a its gradient (both are vector fields on the meshes). You can use these examples to test your implementation.

Task 2 (Mesh Editing).

Implement a tool for editing triangle meshes (a simplified version of the brushes tool we discussed in the lecture). It should allow to specify a 3×3 matrix A, which is applied to the gradient vectors of all selected triangles of the mesh. Then, the vertex positions of the mesh are modified such that the gradient vectors of the new mesh are as-close-as-possible (in the least-squares sense) to the modified gradients.
Remark: The vertex positions is only determined up to translations of the whole mesh in $\mathbb{R}^3$. You can deal with this by keeping the barycenter of the mesh constant.

Task 3 (Summary of Experiences).

Write one short user manual for your tool. Apply your tool to deform 3D-meshes and write a short summary of your experiences (2-3 pages including images).

Hints.

Concerning the construction of the sparse matrix G:

- Use the `class jvx.numeric.PnSparseMatrix`
- Constructor: `PnSparseMatrix(`$n, m, 3$`)`, where n equals 3∗the number of faces of the mesh and n equals the number of vertices of the mesh. The last argument reserves space for storing 3 entries per row. Alternatively, you could use `PnSparseMatrix(`n, m`)`. Then the space for the entries will be allocated when you set them.

- You can use the method `addEntry(int` k`, int` l`, double` $value$`)` for constructing the matrix. The method adds $value$ at position k, l in the matrix. If the matrix entry with k, l does not exist, the space for storing it is created.

For multiplication of sparse matrices you can use:
$AB =$`PnSparsematrix.multMatrices`$(A, B, null)$;

For multiplication of a sparse matrix and a vector you can use:
$Av =$`PnSparsematrix.rightMultVector`$(A, v, null)$;. This method generates an new PdVector that is the product of the matrix and the vector. If a PdVector w for storing the result is already allocated, use
`PnSparsematrix.rightMultVector`(A, v, w);. The method then additionally returns a reference to w.

For solving the sparse linear systems (Task 2), you can use
`dev6.numeric.PnMumpsSolver`. This class offers an interface to the direct solvers of the MUMPS library. To solve the system $A\,x = b$, you can use the method `solve(A, x, b, PnMumpsSolver.Type.GENERAL_SYMMETRIC)`.
For solving a number of systems with the same matrix, compute the factorization once using:

- `public static long factor(PnSparseMatrix matrix, Type sym)`

and use

- `public static native void solve(long factorization, PdVector x, PdVector b)`

for solving the systems.

The MUMPS library should work on WINDOWS 64-bit systems. For MAC, I added the file `libMumpsJNI.jnilib` to blackboard. Please copy the file to the "dll"folder. This file may not work on your MAC, because it depends on other libraries including `gcc` and `gfortran`.

If the MUMPS library does not work on your system, you can use
`jvx.numeric.PnBiconjugateGradient` instead. However, this is less efficient (do not use too large meshes in this case).