# Practical Assignment 2
# (Differential Coordinates and Shape Editing)

Andrea Alfieri - 5128315
Blanca Guillen Cebrian - 5157099
Panagiotis Reppas - 5141435

June 3th 2020

## 1 Algorithm Description

### 1.1 Task 1

The code for this task can be found under `workshop/Task1_IP.java` and `workshop/Task1.java`. We implemented an algorithm that computes the sparse gradient matrix G, combinatorial Laplace matrix L, contangent matrix S and the mass matrix M. The code also tests the correct computation of these matrices. In addition, we created an interface with 4 buttons that the user can press depending on the matrix he wants to calculate.

After clicking on a button of the interface, the corresponding function of `Task1` is called and a `PnSparseMatrix` is returned. The computation of the geometrical Laplace matrix is also possible by using the function `matrixL(Matrix_M, Matrix_S)`. This class contains some methods to print different types of variables. For the case of a `PnSparseMatrix`, instead of printing all the components, which would not be manageable, we print the dimension of the matrix and the number of non zero elements in the matrix.

#### Matrix G

To compute this matrix, the function `matrixG()` is called, which returns it as `PnSparseMatrix` object at the end of the computation. First, the sparse matrix is initialized with dimensions $(3m, n)$, where $m$ is the number of triangles in the mesh and $n$ is the number of vertices.

For each triangle we calculate the 3x3 local gradient matrix. Then, we sort the entries to the corresponding locations in the global matrix as follows:

- row = 3*index of the triangle
- column = index of the vertices of the triangle

### Matrix L

The matrix $L$ has dimension $(n, n)$, where $n$ is the number of vertices in the mesh, and is initialized as having all zeros. For each vertex $i$ in the mesh, we first find the list of its neighbours. This search is done by iterating over all triangles and looking for those that contain such vertex. Therefore, the neighbors are all the points of the triangles that contain the original vertex. $N(i)$ is the list of neighbours of $i$.

Then, we compute $deg(i)$ = number of neighbours of $i$, and set the matrix elements as follows:

$$L_{ij} = -\frac{1}{deg(i)} \quad \text{for } j \in N(i)$$

$$L_{ii} = -\sum_{j \in N(i)} L_{ij}$$

The entire computation is done once for each vertex. In other words, this algorithm is $O(n \cdot m)$, where $m$ is the number of triangles. However, testing shows that it only becomes slow with very large meshes.

### Matrix S

Matrix $S$ is computed as the product of $G^{\mathrm{T}} \cdot M_V \cdot G$, where $G$ is the matrix described above and $M_V$ is a diagonal matrix of dimensions $(3m, 3m)$ ($m$ is the number of triangles in the mesh). The resulting multiplication is a matrix of dimensions $(n, n)$.

$M$ is computed by initializing it to all zeros and then setting the elements $M_V(i, i) = M_V(i + 1, i + 1) = M_V(i + 2, i + 2) =$ area of triangle $i$. In other words,

$$M_V = \begin{pmatrix} A_{T_1} & 0 & \cdots & 0 \\ 0 & A_{T_2} & & \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & A_{T_m} \end{pmatrix}$$

$$A_{T_i} = \mathrm{diag}(\mathrm{area(i)}, \mathrm{area(i)}, \mathrm{area(i)})$$

Again, these areas are computed by going through all the triangles once, but requires the computation of the matrix $G$ beforehand.

### Matrix M

$M$ is a diagonal matrix with dimension $(n, n)$, where $n$ is the number of vertices of the mesh. This is simply computed as:

$$M_{ii} = \frac{1}{3} \sum_{t \in N_t(i)} A(t)$$

where $N_t(i)$ is the list of triangles that contain vertex $i$ and $A(t)$ is the area of the triangle. The computation of this matrix is very similar to that of matrix

$L$ since, for each vertex $i$, we search over all triangles for those that contain $i$, keeping track of the total sum of their area and finally dividing it by 3.

## 1.2 Task 2

The code for this task can be found under `workshop/Task2_IP.java` and `workshop/Task2.java`. For the purposes of this task we implemented two brush tools for editing triangle meshes. In both cases the user highlights a subset of the mesh to deform and specifies in the interface a 3x3 matrix $A$ used for the deformation.

To begin with, we store the vertex coordinates of all vertices in three different vectors $v_x, v_y, v_z$. This is followed by the computation of the $G$ matrix the same way as described before, which is used to compute the three vectors $g_x, g_y, g_z$ as follows:

$$g_x = G \cdot v_x \quad g_y = G \cdot v_y \quad g_z = G \cdot v_z$$

When using Laplace coordinates, the above equations are the same, where the only difference lies in the fact that we use the Laplace coordinates $\delta_x, \delta_y, \delta_z$ instead of $v_x, v_y, v_z$. The Laplace coordinates are computed as follows:

$$\delta_x = L \cdot v_x \quad \delta_y = L \cdot v_y \quad \delta_z = L \cdot v_z$$

by using the same algorithms for the computation of L described for task 1.

The gradient vectors are of dimension $3m$ and are modified based on which triangles are selected by the user. If triangle $i$ is selected, we retrieve its gradient vector from each of the three directions and multiply these by $A$, the deformation matrix. These final vectors

$$\tilde{g_x}, \quad \tilde{g_y}, \quad \tilde{g_z}$$

are the modified versions of the original $g$ vectors, with the deformed coordinates.

Finally, by solving the following linear systems for $v$ we get the coordinates of the vertices of the deformed mesh.

$$\begin{cases} G^T M_V G v_x = G^T M_V \tilde{g_x} \\ G^T M_V G v_y = G^T M_V \tilde{g_y} \\ G^T M_V G v_z = G^T M_V \tilde{g_z} \end{cases}$$

Matrix $M_V$ can be calculated the same way as in Task 1 (matrix $S$). The rest of the elements are known and we only need to calculate $v_x, v_y, v_z$. Since we know every other element it is easy to make the multiplications and bring the system to a form similar to:

$$Ax = b$$

In order to solve these systems we use the PnConjugateGradientMatrix() function. For the second brush tool, the solutions come from different systems:

$$\begin{cases} G^T M_V G L v_x = G^T M_V \tilde{g_x} \\ G^T M_V G L v_y = G^T M_V \tilde{g_y} \\ G^T M_V G L v_z = G^T M_V \tilde{g_z} \end{cases}$$

# 2 User Interface and how to run the code

## 2.1 Task 1

The interface for this task is shown in figure 1, which contains 4 buttons. Each of them computes the requested matrix for the selected mesh. Moreover, these buttons automatically run different tests after the computation. Finally, if all tests are passed, we print the matrix in the console. Since these are all very big sparse matrices, we only print the name, number of rows, number of columns and number of non-zero entries.

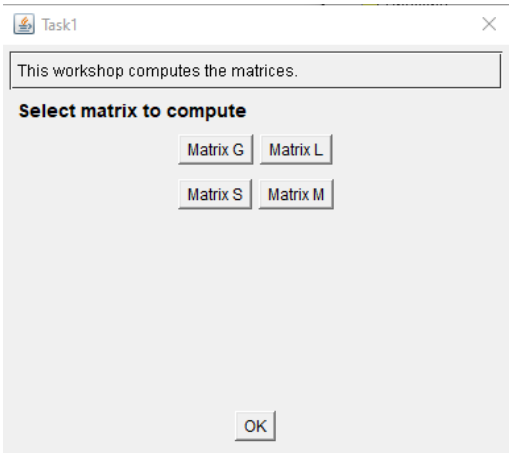



Figure 1: Interface for Task 1

## 2.2 Task 2

The user interface for task 2 is shown in figure 2. It functions as follows: first, the user must choose the area he wishes to edit. This can be done in Javaview by following the path Method - Mark - Mark Elements. After the area is chosen, the user has to insert the values of the 3x3 deformation matrix in the corresponding text boxes. Moreover, the user can also choose the editing tool he wants to use by pressing the right button. Finally, button RUN initiates the algorithm and the new mesh is shown.

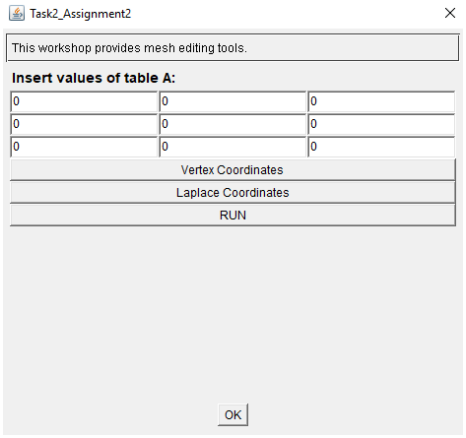



Figure 2: The user interface for task 2

# 3 Tests for correctness of the algorithm

## 3.1 Task 1

For each matrix, different tests were designed to check if the computation is correct.

### Matrix G

- Number of non-zero elements should equal $9 * m$ (number of triangles).
- Check if the global gradient matrix is correctly assembled. To do this, we print the vertices of a triangle $i$, and check if the column numbers of the non-zero elements in the corresponding rows of this triangle $i$ (from $3 * i$ to $3 * i + 3$) equal the indices of its vertices.

In this case, the mesh has 53628 faces. As it can be observed in figure 3, there are 482652 non zero elements, which equals 9*53628. It also shows the vertex array of face 1373 and, below this, the location in the matrix of the non-zero entries in the corresponding rows. The columns where these entries are located coincide with the vertices, so we can conclude that the local matrix was correctly sorted into the global matrix. The same happens for another random face, which in this case is the 3057th.

### Matrix L

- Check if the value of the diagonal entry equals the negative sum of the rest of non-zero entries of the row.

```
Testing matrix G...
Face 1373: [15955, 15500, 14744]
(4119, 14744)
(4119, 15500)
(4119, 15955)
(4120, 14744)
(4120, 15500)
(4120, 15955)
(4121, 14744)
(4121, 15500)
(4121, 15955)
Face 3057: [4900, 5103, 5244]
(9171, 4900)
(9171, 5103)
(9171, 5244)
(9172, 4900)
(9172, 5103)
(9172, 5244)
(9173, 4900)
(9173, 5103)
(9173, 5244)
G matrix:
NUMBER OF ROWS: 160884
NUMBER OF COLUMNS: 26815
There are 482652 non zero elements in the sparse matrix
COMPLETED MATRIX G CALCULATION AND TESTING
```

Figure 3: Test for matrix G.

We can see in figure 4a that this in effect happens. Then, we print as well for some rows the columns where there is a non zero entry. With this test, we can check that the average valence is almost 6.

```
Finished matrix L computation.
(1, 0): -0.14285714285714285
(1, 1): 0.857142857142857
(1, 2): -0.14285714285714285
(1, 4): -0.14285714285714285
(1, 5): -0.14285714285714285
(1, 6): -0.14285714285714285
(1, 8): -0.14285714285714285
(60, 35): -0.16666666666666666
(60, 49): -0.16666666666666666
(60, 60): 0.8333333333333333
(60, 63): -0.16666666666666666
(60, 68): -0.16666666666666666
(60, 75): -0.16666666666666666
Testing matrix L...
```

(a)

```
Testing matrix L...
**********************************************
Row 10: [0, 6, 10, 17, 35, 36, 49]
Row 20: [5, 9, 15, 16, 20, 30]
Row 30: [9, 16, 20, 29, 30, 34, 38]
Row 50: [39, 46, 50, 53, 69, 72]
Row 1000: [854, 923, 961, 994, 1000, 1173]
Row 6000: [5489, 5737, 5947, 6000, 6090, 6149, 6292]
**********************************************
L:
NUMBER OF ROWS: 26815
NUMBER OF COLUMNS: 26815
There are 187699 non zero elements in the sparse matrix
COMPLETED MATRIX L CALCULATION AND TESTING
```

(b)

Figure 4: Test for matrix L.

**Matrix S**

- Check if the value of the diagonal entry equals the negative sum of the rest of non-zero entries of the row.
- Check if it is symmetric: check if entry (i,j) equals (j,i) for some random points.
- Check if the position of the non-zero elements in matrix $S$ coincides with the position of the non-zero elements in matrix $L$. They must also have the same number of non zero elements.

In this case, to check the first and second test we initialize a boolean variable (`found`) as false that becomes true if there is any error. If the matrix is correct, then it prints the message "MATRIX S IS CORRECT". If there is any error, it would be printed in the console. This is shown in the piece of code in figure 5.

```
boolean found = false;
for(int row : rows_to_check)
    for(int col : cols_to_check)
        if(Math.abs(S.getEntry(row, col) - S.getEntry(col, row)) > 0.001) {
            PsDebug.message("(" + row + ", " + col + ") is different than (" + row + ", " + col + ")");
            found = true;
        }

    double sum = 0;
    for (int col = 0; col < S.getNumCols(); col++)
        if(col != row && S.getEntry(row, col) != 0.0)
            sum += S.getEntry(row, col);

    if(Math.abs((-sum) - S.getEntry(row, row)) > 0.001) {
        PsDebug.message("For row " + row + " the sum is " + (-sum) + " but the diagonal is " + S.getEntry(row, row));
        found = true;
    }
}
```

Figure 5: Piece of code matrix S.

```
Testing matrix S ...
************************************************
Row 10: [0, 6, 10, 17, 35, 36, 49]
Row 20: [5, 9, 15, 16, 20, 30]
Row 30: [9, 16, 20, 29, 30, 34, 38]
Row 50: [39, 46, 50, 53, 69, 72]
Row 1000: [854, 923, 961, 994, 1000, 1173]
Row 6000: [5489, 5737, 5947, 6000, 6090, 6149, 6292]
************************************************
MATRIX S IS CORRECT
S:
NUMBER OF ROWS: 26815
NUMBER OF COLUMNS: 26815
There are 187699 non zero elements in the sparse matrix
COMPLETED MATRIX S CALCULATION AND TESTING
```

Figure 6: Test for matrix S.

For the last test, we have to compare this results with the ones in matrix $L$. As we can see in figures 4a and 6, both the location of the non zero entries and the number of these ones is the same for both matrices, which is consistent with the theory.

As we used $G$ to compute $S$, and we just checked that $S$ is correct, this is also another proof that shows that $G$ is correctly computed.

### Matrix M

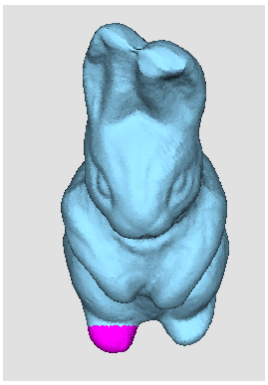
Since matrix M is manually created by providing each element of the diagonal, we just visually inspected it afterwards to check if the numbers were in the correct place by printing them in the console.

## 3.2 Task 2

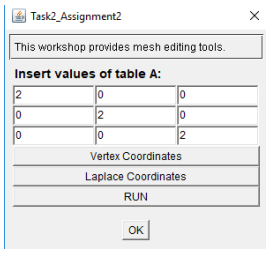
### Vertex Coordinates

In order to check the behaviour of our algorithm, we conducted a number of tests using different input matrices.

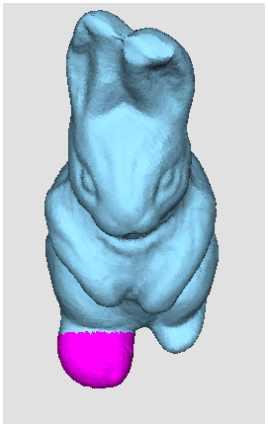
To begin with, we tested the effects of a scale matrix on an object. More specifically, as it can be seen in Figure 7 and Figure 8 we doubled the rabbit's foot and tripled the rabbit head's ears:



(a) Mesh before edit

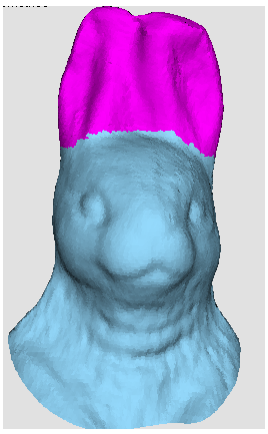



(b) Scale Matrix

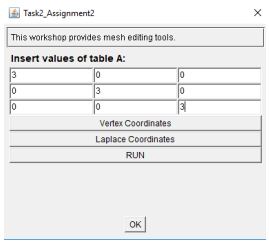


(c) Mesh after scale

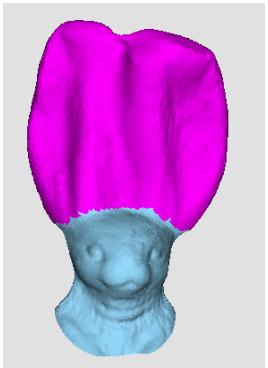
Figure 7



(a) Mesh before edit




(b) Scale Matrix

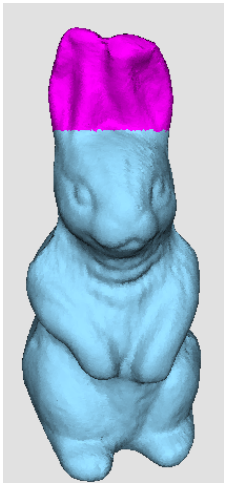


(c) Mesh after scale

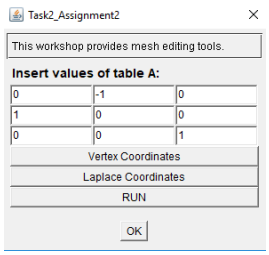
Figure 8

Moreover, we implemented a rotation matrix on the rabbit's ears. The results

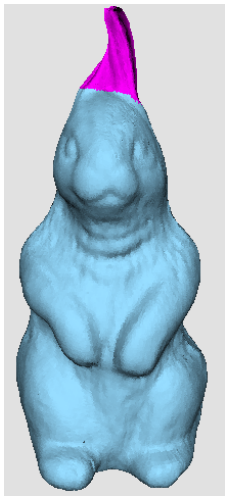
can be seen in Figure 9.



(a) Mesh before edit



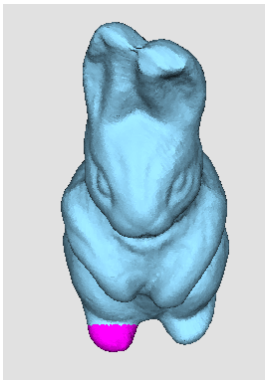

(b) Rotation Matrix



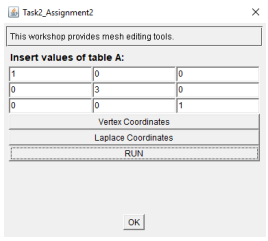
(c) Mesh after scale

Figure 9

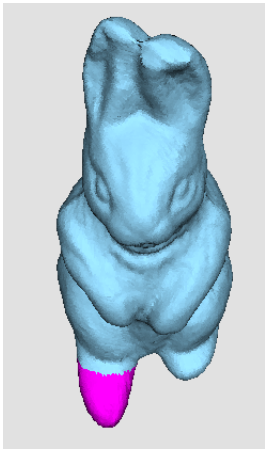
Last but not least, in Figure 10 we present the effects of an anisotropic scale on the rabbit's foot.



(a) Mesh before anisotropic scale




(b) Anisotropic Scale Matrix



(c) Mesh after anisotropic scale

Figure 10

**Laplacian Coordinates**

For the Laplacian coordinates, the algorithm works and does not throw any exceptions, but the systems converge in a much longer time and the final results are not as expected. By this we mean that the object is deformed but there is always some sort of "stretching" happening. To solve this problem we tried to use different solvers but without success.

# 4 Evaluation

## 4.1 Matrix G

The computation of this matrix is $O(m)$, where $m$ is the number of triangles in the mesh. This is because we iterate through all triangles once and the computation inside this for loop is $O(1)$

## 4.2 Matrix L

As stated before, the complexity of this algorithm is $O(n \cdot m)$, where $n$ is the number of vertices and $m$ is the number of triangles. This computation is still quite fast and only becomes slow with very large meshes.

## 4.3 Matrix S

The computation of matrix $M_V$ is $O(m)$, where $m$ is the number of triangles in the mesh. This is because each triangle is addressed once and the calculation of its area is $O(1)$. The computation of $G$ is also $O(m)$ as shown before. Therefore the computation of $S$ is $O(2m) = O(m)$.

## 4.4 Matrix M

The computation of $M$ is also $O(n \cdot m)$ because, similarly to matrix $L$, for each vertex we search through all the triangles for the ones that contain such vertex. The internal computation is then to just sum all the areas of these triangles, which is an operation of constant complexity $O(1)$.

## 4.5 Task 2

The majority of the complexity of this task lies in the system solvers we used. In fact, the processing steps that come before asking the solvers for a solution are just $O(m_s) + O(m) = O(m)$, where $m_s$ is the number of selected triangles and $m$ is the total number of triangles. The complexity of a solver that uses direct methods is $O(n^3)$, which is generally shrunk to $O(n^2)$ by using iterative methods (like Krylov ones). With sparse matrices, these algorithms can converge in $O(n)$, but judging by the time our solvers are taking for the models we tested,

we decided to use the upper bound of $O(n^2)$. Therefore, the total complexity of this task is $O(m) + O(n^2) = O(n^2)$.

For this task, we also tried to evaluate how much a simpler model affects the speed and accuracy of the algorithms. We used the same models describe in the figures above but applied the "Simplification" effect by reducing the number of faces by 75%. This really improved the speed of the task but the results were still very accurate.

# 5 Division of labor

- Task 1: Andrea Alfieri and Blanca Guillen Cebrian.
- Task 2: Andrea Alfieri and Blanca Guillen Cebrian.
- Report and tests: Andrea Alfieri, Blanca Guillen Cebrian and Panagiotis Reppas.