COMP0119 Acquisition and
Processing of 3D Geometry, 22/23
Assignment 1

01/02/23

1 Point-to-point ICP and weighted ICP

1.1 Implementation and results of our point-to-point ICP algorithm

We will use MeshLab as our mesh viewer for this assignment. By looking at the models we are given, we will use bun000 as our model M_1, and bun045 as our model M_2 because they look almost aligned and have a large overlap.
To implement the point-to-point ICP algorithm, we have followed the methodology seen in class i.e

1. Constructing the P and Q sets from roughly aligned models (here, we have chosen bun000 and bun045)
2. Selecting a subset of points p_i

   This is the subsampling set. For performance reasons, we have chosen to use 10000 points, knowing that the models are comprised of around 40000 points. We will see the effects of subsampling later in question 4. At the moment, we subsample P and Q by the same amount of points.
3. Assigning the $Q \rightarrow P$ correspondence

   In the point-to-point version, we just iterate over each P point, and look for its closest neighbor. It is done by querying the k-d tree (from the sklearn package) built from the Q set, so as to speed computation.
4. Computing $\mathbf{R}$, $\mathbf{t}$.

   We do so by computing the closed-form solution derived from the alignment error equation $E(\mathbf{R}, \mathbf{t}) = \sum_i \|\mathbf{R}\mathbf{p_i} + \mathbf{t} - \mathbf{q_i}\|^2$.
5. Moving the Q points according to our transform matrices : $Q \rightarrow \mathbf{R}Q + \mathbf{t}$
6. At this point, either we go back to step 3, or we exit the loop. To make this choice, we have implemented two choices: an epsilon precision and a maximum number of iterations. Either the algorithm stops because the residue between the previous alignment and the new one is lower than epsilon, or it stops because we have run over the maximum number of iterations.

Running the algorithm on our models, for a maximum number of 100 iterations, an epsilon tolerance of $1e-3$, we obtain the following alignment (see Figures 1-3):

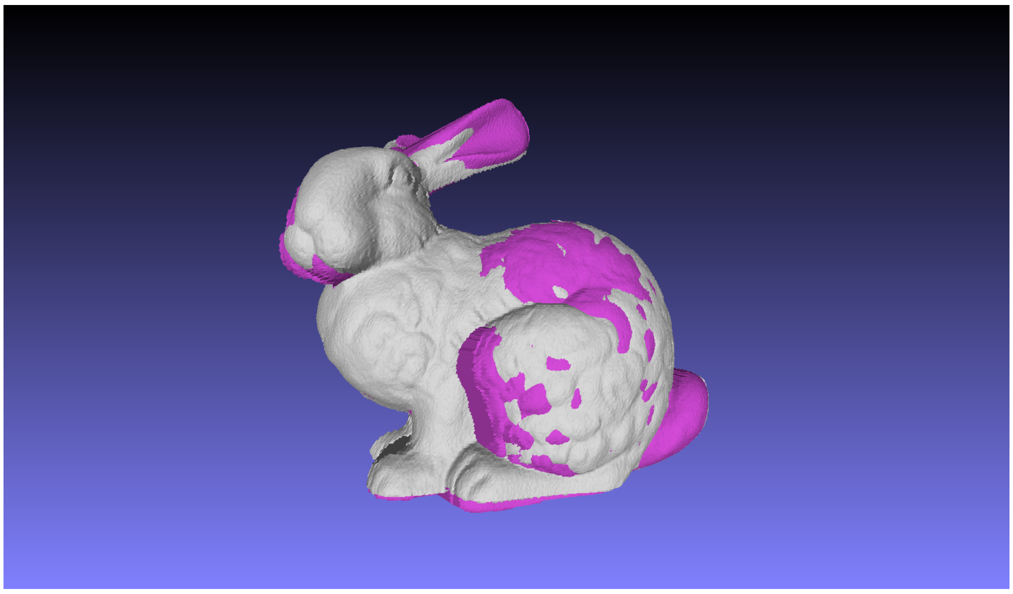

Figure 1: Alignment of bun000 with bun045, front view (bun000 in white, aligned bun045 in pink)

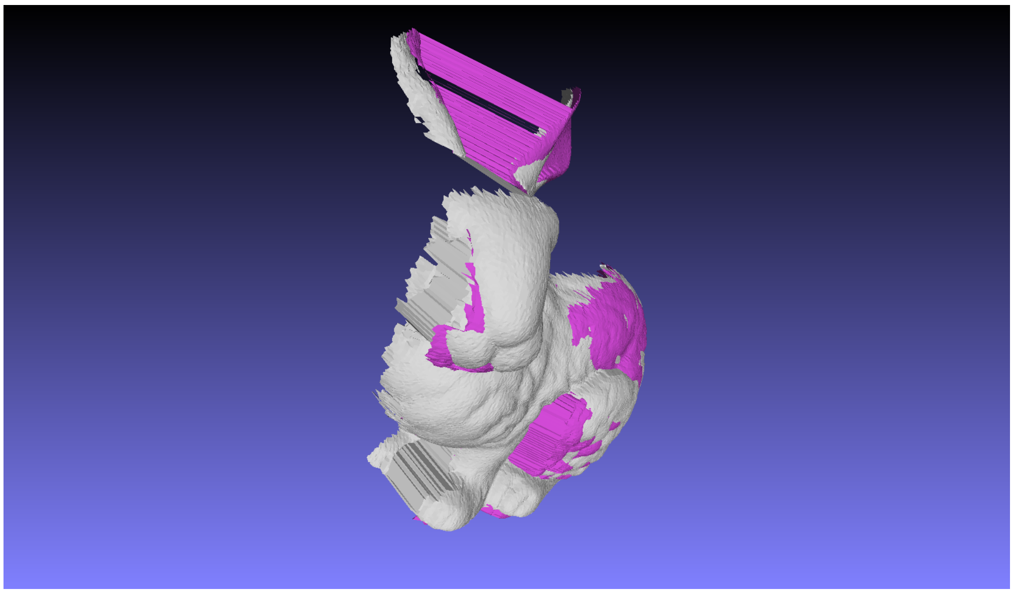

Figure 2: Alignment of bun000 with bun045, side view (bun000 in white, aligned bun045 in pink)

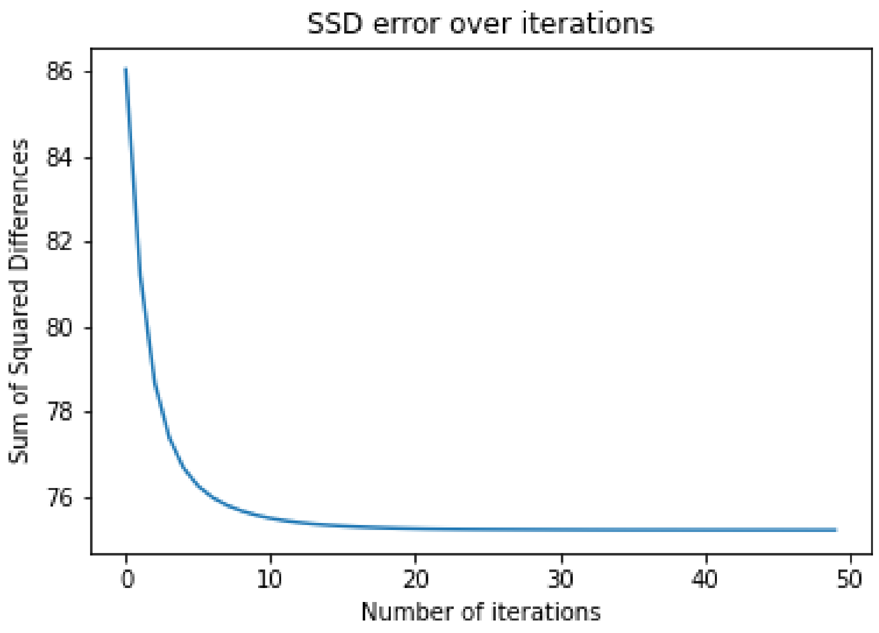

Figure 3: Sum of squared differences of the P and Q points over the iterations

The algorithm converges in about 20 iterations. The result is satisfactory. Let's remember we have started with two almost aligned models. In the rest of the coursework, we will investigate different situations where the algorithm is put to the test.

1.2 Deriving the weighted ICP

We are now asked to derive a weighted version of the ICP, where each point from $P = (p_i)$ is assigned a weight w_i :

$$E(\mathbf{R}, \mathbf{t}) = \sum_i w_i \|\mathbf{p_i} - \mathbf{R}\mathbf{q_i} - \mathbf{t}\|^2.$$

We will follow the methodology seen in class. We have also adopted the notation seen in class. It gives the same result as the one asked in the coursework sheet by symmetry of P and Q, and the fact that $\|a-b\|^2 = \|b-a\|^2$ To minimize this sum, we start by looking at the derivative of E with respect to t, and set it to 0 (because we are looking for a minimum):

$$\sum_i 2 * (w_i \mathbf{p_i} - w_i \mathbf{R}\mathbf{q_i} - w_i \mathbf{t}) = 0$$
$$\implies \sum_i w_i \mathbf{p_i} - \sum_i w_i \mathbf{R}\mathbf{q_i} - \sum_i w_i \mathbf{t} = 0.$$

$\mathbf{t}$ being independent of i, we can say that $\sum_i w_i \mathbf{t} = \mathbf{t}(\sum_i w_i)$, and divide everything by the sum of all weights to retrieve $\mathbf{t}$ in order to be able to replace it in the subsequent equations. We retrieve equation (5) from the derivation sheet, but with new definitions for $\bar{\mathbf{p}}$ and $\bar{\mathbf{q}}$:

$$\bar{\mathbf{p}} = \mathbf{R}\bar{\mathbf{q}} + \mathbf{t}, \qquad \bar{\mathbf{p}} = \frac{\sum_i w_i \mathbf{p_i}}{\sum_i w_i}, \bar{\mathbf{q}} = \frac{\sum_i w_i \mathbf{q_i}}{\sum_i w_i}.$$

This makes sense, because before using the weighted version, we divided everything by n (i.e the number of points), which is actually the weighted version but considering an equal weight for every point i.e $\frac{1}{n}$. We retrieve the unweighted version using $w_i = \frac{1}{n} \ \forall i$.
We still have some computation to do not to forget other w_i terms. Replacing this new expression of $\mathbf{t}$ in E, we obtain:

$$E(\mathbf{R}, \mathbf{t}) = \sum_i w_i \|\mathbf{p_i} - \mathbf{R}\mathbf{q_i} + \mathbf{R}\bar{\mathbf{q}} - \bar{\mathbf{p}}\|^2$$
$$\implies E(\mathbf{R}, \mathbf{t}) = \sum_i \|\sqrt{w_i}(\mathbf{p_i} - \bar{\mathbf{p}}) - \mathbf{R}\sqrt{w_i}(\mathbf{q_i} - \bar{\mathbf{q}})\|^2.$$

Once again, by using the same notation and modifying the expression, we can write:

$$E(\mathbf{R}) = \sum_i \|\tilde{\mathbf{p_i}} - \mathbf{R}\tilde{\mathbf{q_i}}\|^2, \qquad \tilde{\mathbf{p_i}} = \sqrt{w_i}(\mathbf{p_i} - \bar{\mathbf{p}}), \tilde{\mathbf{q_i}} = \sqrt{w_i}(\mathbf{q_i} - \bar{\mathbf{q}}).$$

After this, we have nested the weights into our new points, therefore the remainder of the proof is exactly the same as for the unweighted ICP. We finally end up with:

$$\mathbf{R} = \mathbf{V} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(\mathbf{V}\mathbf{U^T}) \end{pmatrix} \mathbf{U^T}, \qquad \mathbf{t} = \bar{\mathbf{p}} - \mathbf{R}\bar{\mathbf{q}}.$$

where

$$\bar{\mathbf{p}} = \frac{\sum_i w_i \mathbf{p_i}}{\sum_i w_i}, \quad \bar{\mathbf{q}} = \frac{\sum_i w_i \mathbf{q_i}}{\sum_i w_i}, \quad \tilde{\mathbf{p_i}} = \sqrt{w_i}(\mathbf{p_i} - \bar{\mathbf{p}}), \quad \tilde{\mathbf{q_i}} = \sqrt{w_i}(\mathbf{q_i} - \bar{\mathbf{q}}), \quad SVD(\sum_i \tilde{\mathbf{q_i}}\tilde{\mathbf{p_i}}^T) = U\Sigma V^T.$$

2 Rotation and misalignment

In this part, we rotate the initial mesh along an axis, to look at how the algorithm manages to find back its marks when the initial alignment is not what is expected.
To do so, we have created an array of angles to try (that we have converted in radians), and we initialize the same mesh twice: one is the mesh unmodified, the other is the mesh rotated by the angle. We perform the ICP algorithm on those meshes, and we take a look at the number of iterations to converge, and the alignment error (see Figure 4)

We can see that the more the mesh is rotated, the more iterations the algorithm needs to retrieve the initial alignment. The error metric seems to stay stable though. But one problem that can arise is to have a ”wrong” convergence i.e that the algorithm seems to converge, but ends up producing a completely wrong alignment. For instance, if we take the 180° rotation, how can we expect our algorithm to know that we have twice the same model, but the second is upside-down? The ICP algorithm only looks for the closest points to make them stitch, it does not learn. From our point of view, it seems natural to rotate back the bunny in the right way, and then stitch them. The algorithm cannot see the different features of the model (e.g ears, paws) to establish a first alignment, and then proceeds as usual. As a result, it will stitch the closest points given the initial alignment, and will only try to minimize the score starting from here.

As we can see (see Figures 5-6), the algorithm is really dependent on that initial alignment. Even the 24° rotation produces a mediocre alignment (the algorithm has stopped because it has reached its desired precision, we must have fallen into a local minimum), the algorithm just tries to stitch the parts that look like they could be roughly aligned. We can conclude from this part that the initial assumption that ”the models are roughly aligned [...] and have a large overlap” is crucial for the validity of the ICP algorithm.

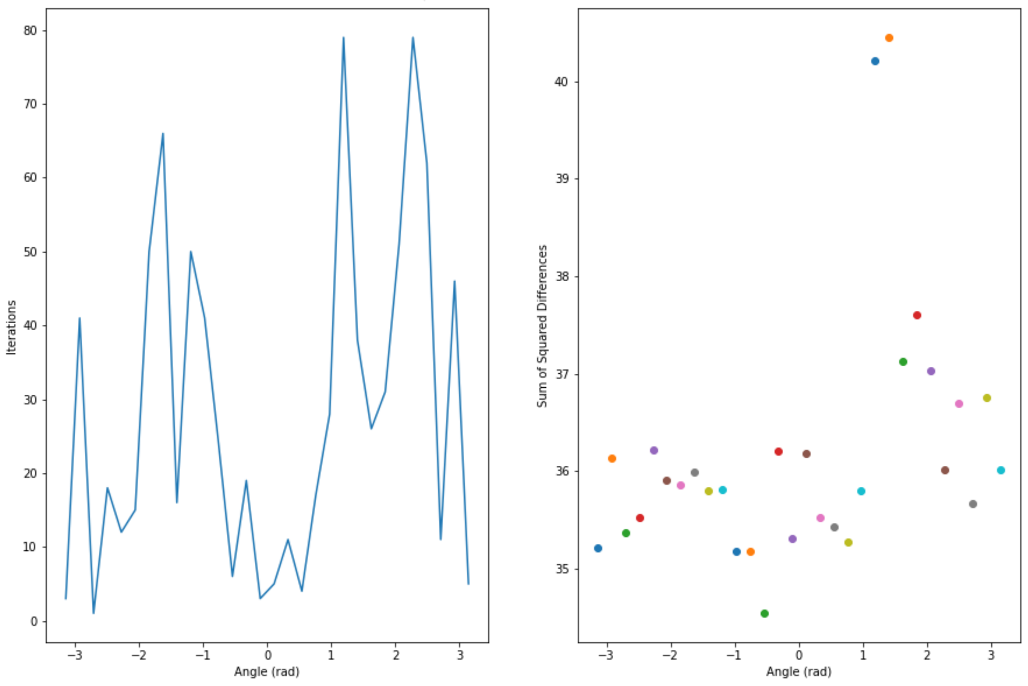

Figure 4: Number of iterations to reach the desired precision (left) and value of the sum of squared differences (right) with respect to the angle of rotation (in radians)

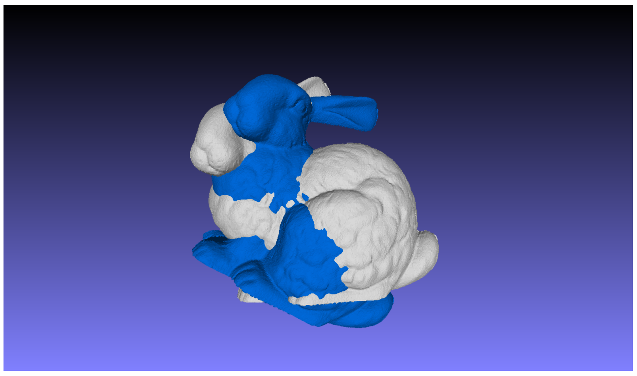

Figure 5: bun000 (white) and bun000 rotated of 24 degrees (blue) before the alignment

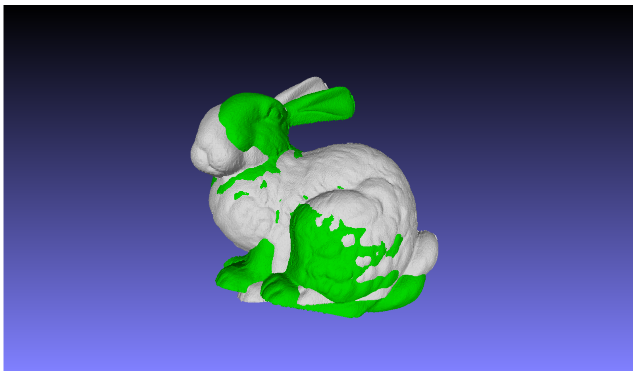

Figure 6: bun000 (white) and bun000 rotated of 24 degrees (blue) after the alignment

3 Noisy models

To compare the different noisy models, we proceed in the same way as the rotated meshes. We add Gaussian noise to the vertices of the M_2 mesh, with a standard deviation each time greater. Here, we see the standard deviation more like a ratio of noise added to the mesh, because we have already scaled the noise to the bounding box of the mesh by multiplying the noise sample by the maximum of each coordinate (e.g for the x-axis: we generate a Gaussian noise sample of mean 0 and variance 1, we multiply it by the greatest (in absolute value) x-coordinate of the vertices, and finally we multiply it by a ratio).
Here are the results below, with some screenshots to look at the alignment of the noisy meshes (see Figures 7-9):

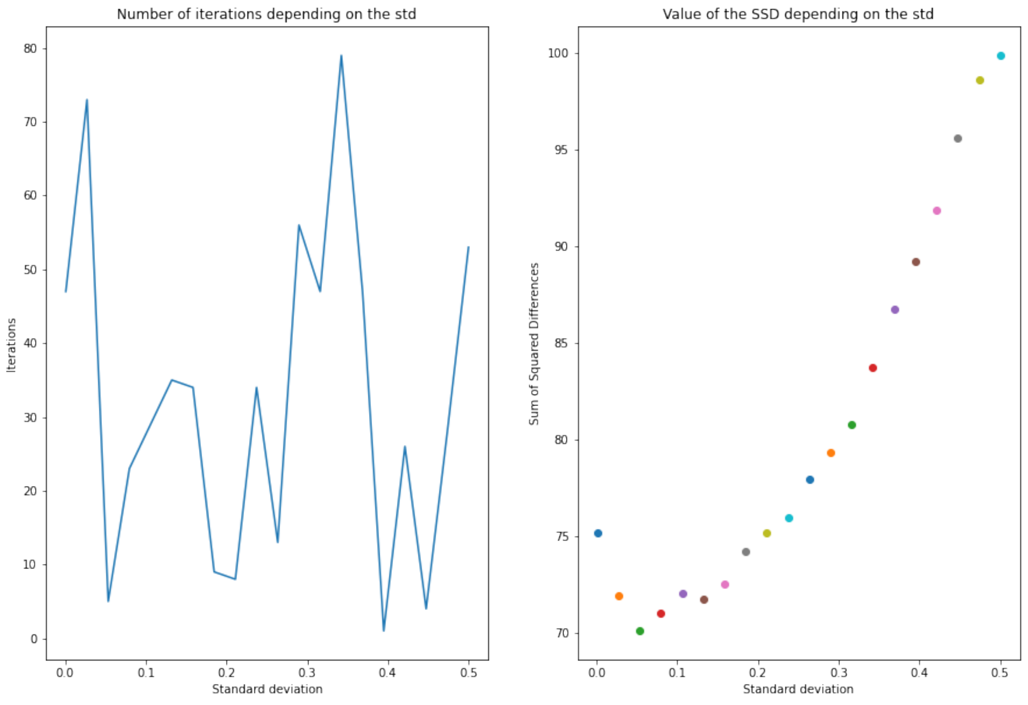

Figure 7: Number of maximum iterations until convergence (left) and sum of squared differences (right) with respect to the variance ratio of the added noise

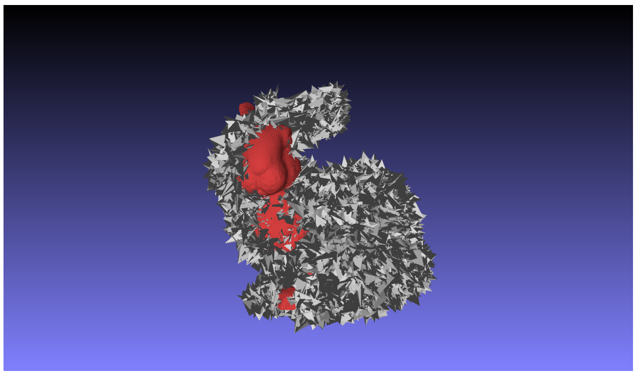

Figure 8: Alignment of bun000 (red) with the noisy bun000 (white), front view

Note: we could have averaged our results over multiple iterations of our method to obtain smoother results.

As expected, the error metric being computed on the distance between P and Q, the noisier the mesh is, the further the points are from the center of the coordinate system, and so the more distance there is between the two point sets. The number of iterations matters less, because if the points are far from each other, the rigid

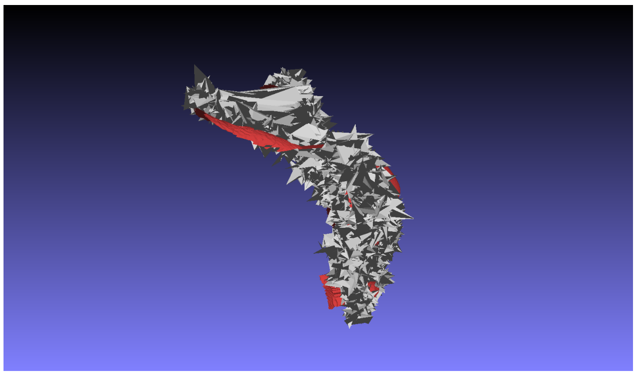

Figure 9: Alignment of bun000 (red) with the noisy bun000 (white), side view

transform cannot apply any scale to them (only rotations and translations are possible in our model), and the best score is achieved regardless of the alignment (in the end, we just try to optimize the distance between a Gaussian cloud and M_1, which makes no sense when trying to have two meaningful meshes overlap).
Though, the Gaussian noise can act as a "scaler", which is an operation that the rigid transform cannot perform. To add a little amount of noise might therefore result useful, if we know exactly our models beforehand.

4 Subsampling

To this point, we have worked with the highest number of points possible (for computational reasons, we cannot use the $\geq$ 40.000 points). Now, let's make the number of sampled points from M_1 or M_2 vary. Intuitively, it seems reasonable to try to sample less points from Q, because those are the points that we are trying to move, and even though we might lose a bit of precision and miss some details, we can still make the algorithm converge over multiple epochs. Conversely, it seems unreasonable to sample less points from P, because those are our "reference" points. If we do this, we do not "miss" some details, we simply erase the existence of these details, possibly resulting in a wrong alignment. Following the method of the last questions, we perform the ICP algorithm for an increasing number of subsampling points, for three cases: number of points from M_1 fixed to a high number and number of points from M_2 varying, number of points from M_2 fixed to a high number and number of points from M_1 varying, number of points from M_1 and M_2 both varying.
We have to define a common metric, because before we assumed the same number of points for M_1 and M_2, but it is no longer the case here. We will take the minimum number of vertices between M_1 and M_2 (so that we can have all the maximum number of points possible for both meshes).

Following our results, we can observe that the best way of aligning two meshes is by sampling uniformly both meshes (see Figure 10).

5 Global alignment

Before proceeding to the global alignment part, we absolutely need to roughly align our models. We have seen in the previous parts that otherwise the results would just make no sense. Therefore, we have rotated in a more convenient position the different models in Meshlab, so that the ICP could produce something correct.
We have implemented two methods:

1. The first one starts by stitching two models together, and then concatenates them together to create a new model. Then, we stitch the third model to the concatenated one, and once the alignment is made, we concatenate the third with the previous ones. It is the idea expressed in the slide 153 of the Registration lecture: we distribute the error between all scans. We progressively build the alignment, by concatenating each new aligned part to the initial model.

   But, we know that each model has a different initial alignment. To prevent any error from spreading, we create a list of permutations, corresponding to the order in which we concatenate the models (i.e we can have [2, 1, 3, 4, 5, 6] which means that we start by aligning 1 to 2, then 3 to (1+2), then 4 to (1+2+3) etc...). We have 6 models, so we have 6!=720 permutations possible. In the end we look at the alignment error for each permutation and we choose the best sequence of alignment.

2. Our second method will not build an alignment mesh by mesh, but rather create a global mesh made of 5 models, and try to align the 6th one to the global one i.e aligning one mesh while fixing the others. We

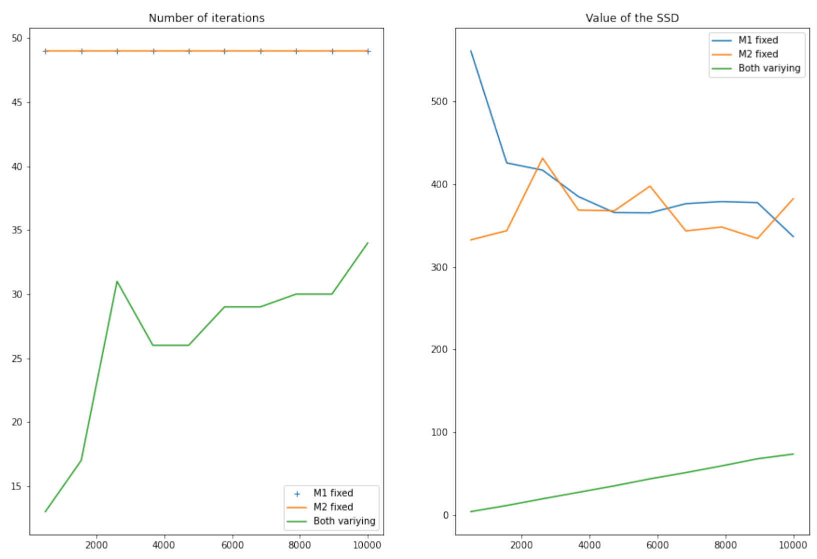

Figure 10: Number of iterations (left) and sum of squared differences (right) with respect to the number of points used for sampling the P and/or Q sets

choose the mesh to align by shuffling the list of meshes. Then, we can perform multiple epochs of this algorithm, so that the meshes progressively converge to a global alignment between them.

The two methods are based on two different principles. The first one tries different sequences to define the best order in which to perform the global alignment. The second one does the initialization randomly, and rather uses a progressive alignment over a given number of epochs. In the end, we could say that the best result can be obtained by combining the two: we first start by estimating the best sequence to initialize the second algorithm with the first algorithm. Then, once we have the best initial alignment possible, we refine our global alignment with the second algorithm.
Unfortunately, the alignment is wrong with our algorithms, it does not produce satisfactory results.

6 Point-to-plane ICP

To build this new version of the ICP algorithm, there are two changes to perform:

1. We have to compute the normals for the P points. It is done using the PCA normal estimation function seen in class during Workshop 1. We basically take the neighboring points of the point at which we want to compute the normal (we also use a k-d tree to do so) and we build the covariance matrix of the (stacked) coordinates of its neighbors. The eigenvectors of this covariance matrix will give us directions of change in coordinates. The eigenvector corresponding to the smallest eigenvalue will be the normal direction, because it is the direction of least change (we are going from a surface, to a non-surface).

2. We have to change the metric from point-to-point $(\mathbf{p_i} - \mathbf{q_i})$ to point-to-plane $(\mathbf{p_i} - \mathbf{q_i})^T \mathbf{n_i}$ where $\mathbf{n_i}$ represents the normal to the plane on which p_i sits (that we have computed at the previous step). This metric enables us to take into account some places where we would rather slide along the surface than just stick each surface to one another.

In the end, we achieve a better score (see Figures 11-14), which probably means that there has been a smoothing in some parts where surfaces have aligned better than with the point-to-point version. But the gain is not huge, because our bunny does not have many flat surfaces that could lead to a wrong alignment. The computation time is a bit higher than the point-to-point version because we have the extra step of computing the normals for P, but also because when estimating the point-to-plane distances, we do not consider only one point, but multiple neighbors (this is a parameter we can adjust). We can guess that the use of this algorithm or the other will depend on the meshes and their characteristics, we might even need to use a combination of both depending on the area of the meshes.

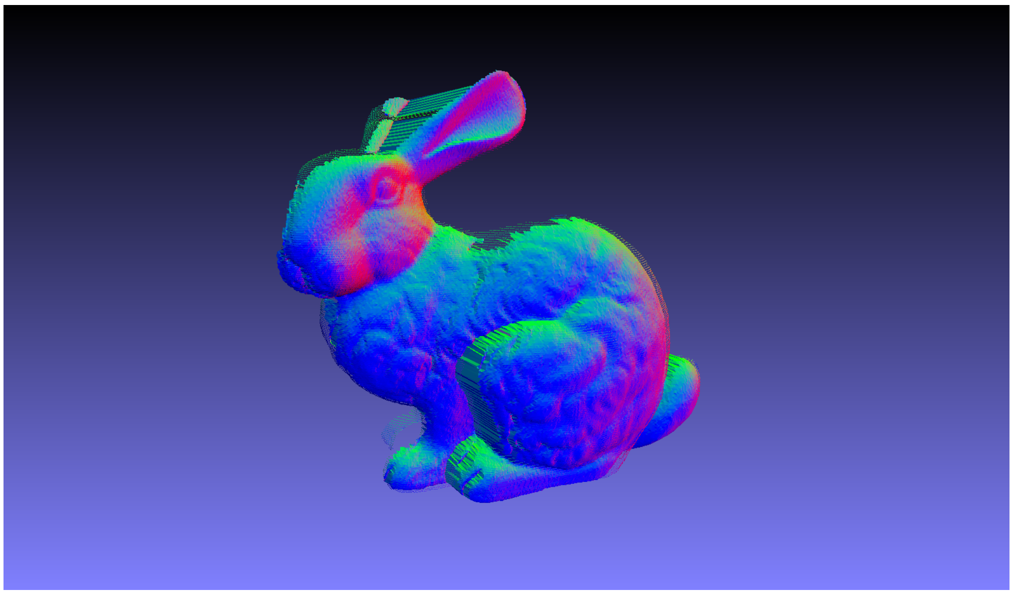
Figure 11: Alignment of bun000 with bun045, front view (bun000 in points, aligned bun045 as the full mesh), the colours come from the map of normals

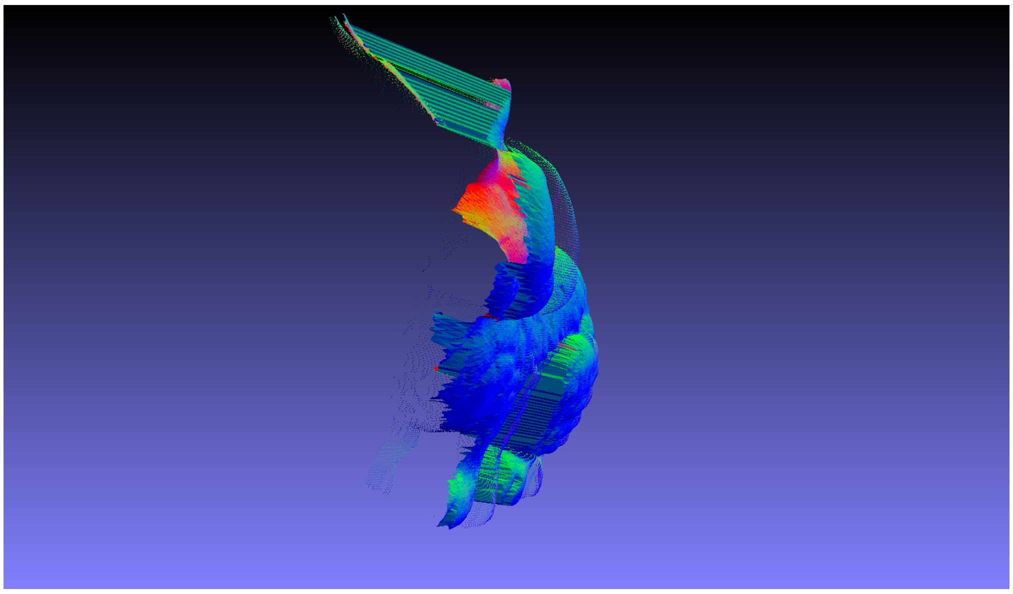
Figure 12: Alignment of bun000 with bun045, side view (bun000 in points, aligned bun045 as the full mesh), the colours come from the map of normals

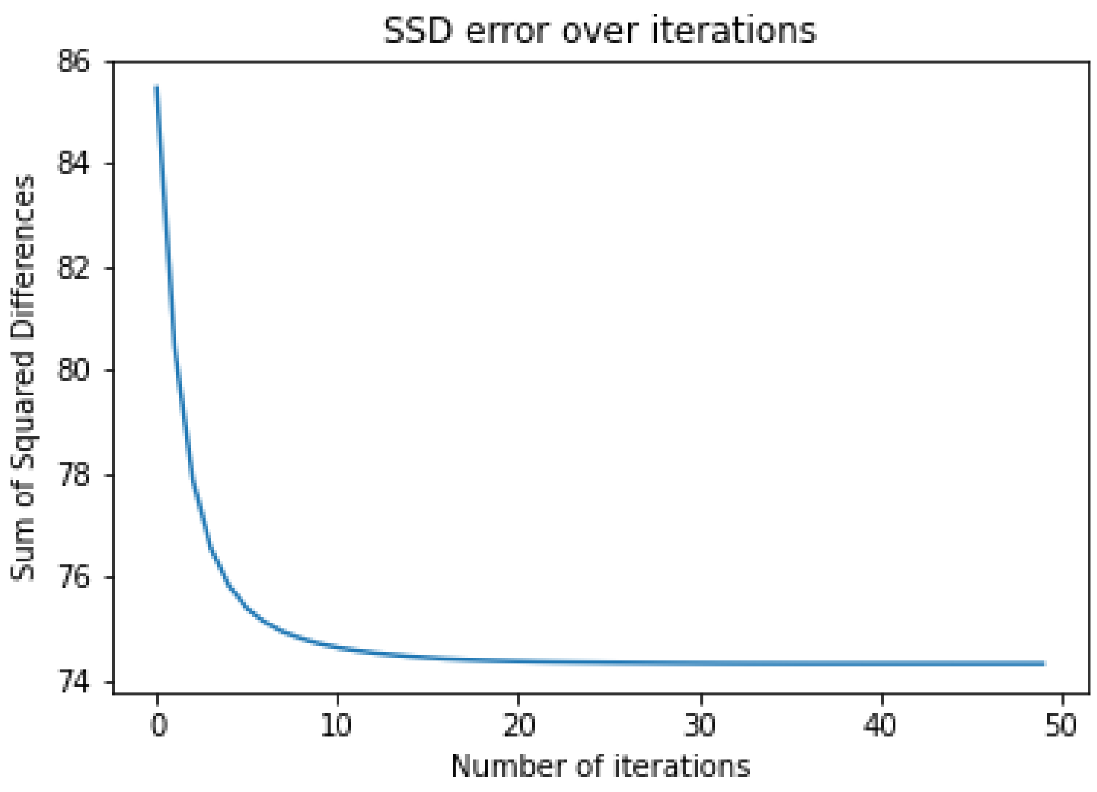

Figure 13: Sum of squared differences of the P and Q points over the iterations

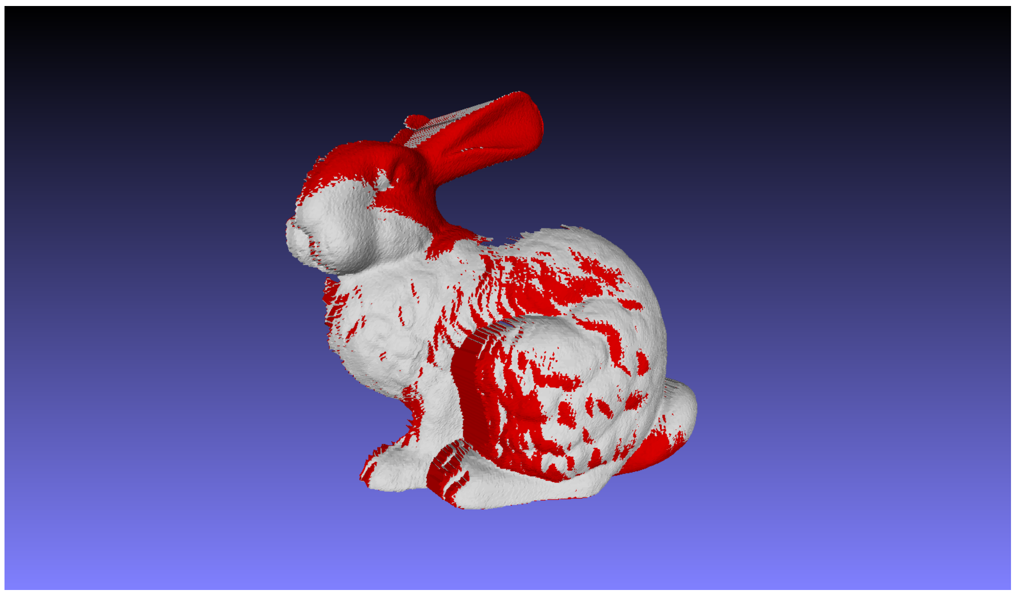

Figure 14: Comparison of the point-to-point alignment (white) with the point-to-plane alignment (red)