3D DATA PROCESSING

LAB 3

Giuseppe Labate mat.2095665

Topic: Iterative Closest Point Cloud Registration
Goal: Given a source and a target point cloud roughly aligned, find the fine alignment transformation of the source to the target cloud.

1) Work description

Task 1 – `find_closest_point`

For this task, the objective was to find the nearest `target_point` for each `source_point`. To do so, I used a KD-tree to store the `target_` point cloud, which accelerates the process of finding the nearest `source_point`.
Next, a `source_clone` point cloud was created to save the `source_for_icp_` point cloud, which is the source point cloud transformed by the current estimation of `transformation_`.

To find the closest point, the `SearchKNN` function with K = 1 was exploited.

When a neighbor is found, its square distance is checked. If it's below a certain arbitrary threshold (0.2 in this case), the source and target indices are saved into two `std::vector` and the mean square error (MSE) is updated to keep track of the current error.

After the for loop is completed, the root mean square error (RMSE) is calculated by taking the square root of the final MSE.

Finally, a tuple composed of the source_indices, target_indices and RMSE is returned.

Task 2 – `get_svd_icp_registration`

Here it was requested to apply the singular value decomposition (SVD) to find the transformation from source to target.

To achieve this, the transformed source point cloud, `source_for_icp_` was saved to source_clone.
The `source_for_icp_` is the initial source point cloud to which the current estimation of the `transformation_` was applied.

Next, the centroids for source and target point clouds were calculated and saved in the `source_centroid` and `target_centroid` 3D vectors, respectively.

With the centroids identified, the difference between each point in the point cloud and its centroid was retrieved, following the nearest neighbor (NN) matching indices order, returned by `find_closest_point`.
In other words, for each iteration ***i***, the `source_clone.points_[source_indices[i]]` and `target_clone.points_[target_indices[i]]` were picked to compute the difference between them and their respective centroids:

```
source_point = source_point - source_centroid;
```

`target_point = target_point - target_centroid;`
applying the following formulas:

$$m'_i = m_i - m_{centroid}$$
$$d'_i = d_i - d_{centroid}$$

Subsequentially, the 3x3 **W** matrix was computed by multiplying the m'_i and the transpose of d'_i and summing the result for each iteration:

$$W \ = \ \sum_{i=1}^{n} m'_i \, {d'_i}^T$$

Note that *m* variables represent for target point and *d* variables represent source points.

The **W** matrix is crucial for finding the transformation between two point clouds.
In fact, the rotation matrix can be written as:

$$\hat{R} = \arg\min_R \sum_{i=1}^{n} \|m'_i - \ Rd'_i\|^2 = UV^T$$

Given:

$$W \ = U\Sigma V^T$$

The U and V matrices were found to derive the rotation matrix.
This was achieved by applying the singular value decomposition (SVD) to **W** by using `Eigen::JacobiSVD<Eigen::MatrixXd> svd(W, Eigen::ComputeFullU | Eigen::ComputeFullV);`

The special case in which the $det(UV^T) \ = -1$ was also handled, by computing the rotation matrix as follows:

$$\hat{R} = U \; diag(1{,}1{,}-1)V^T$$

Note that to check if the determinant was negative, it was needed only to compute and multiply $det(U)$ and $det(V)$ because

$$det(UV^T) = det(U) \times det(V^T) = \ det(U) \times det(V)$$

```
Eigen::JacobiSVD<Eigen::MatrixXd> svd(W, Eigen::ComputeFullU | Eigen::ComputeFullV);

//We have to check if the determinant of the matrix is negative      You, 5 days ago • svd done
//Note that det(U*V^T) = det(U) * det(V^T) = det(U) * det(V),
//so for semplification we can check the product of the determinants
if((svd.matrixU().determinant() * svd.matrixV().determinant()) == -1) //Special reflection case(if corrupted data is present)
{
  Eigen::Matrix3d diag = Eigen::Matrix3d::Identity();
  diag(2,2) = -1;
  R = svd.matrixU() * diag * svd.matrixV().transpose();  //R = U * diag(1, 1, -1) * V^T
}

else //Standard case
  R = svd.matrixU() * svd.matrixV().transpose();  //R = U * V^T

//Now we have to find the translation vector
t = target_centroid - R * source_centroid;   //t = mc - R * dc
```

Figure 1: implementation of SVD

Next, the translation vector was calculated as the difference between the `target_centroid` and the rotated `source_centroid`.

Finally, the rotation matrix and translation vector were concatenated into a 4D transformation matrix, which was then returned.

Task 3 – PointDistance

In task 3 the objective was to complete the `PointDistance` struct to include an auto-differentiable cost function.
This part was similar to the Bundle Adjustment in the previous assignment.
The main difference here was that the function now had to optimize only the transformation array, which is composed by:

- 3 cells for the rotation (expressed in angle-axis representation)
- 3 cells for the translation (tx, ty, tz)

The `operator()` function was used for two purposes:

- to transform the `source_point` first by rotating it with the `AngleAxisRotatePoint` function and then by translating it by exploiting the last 3 cells of `transf` array;
- to retrieve the residual by computing the difference between transformed `source_point` and `target_point` (Note that the residual is 3 dimensional).

Task 4 – get_lm_icp_registration

In this task, it was requested to use the Levenberg-Marquardt algorithm (LM) to find the best roto-translation matrix.

Similar to task 2, the `source_for_icp_` point cloud was stored in `source_clone` to easily access to the point cloud with the current estimated transformation.

Next, for each point in `source_clone`, the cost_function was created using the coupled source and target points as parameters. In the same iteration, the residual block was added to the problem, with the `cost_function` and the `transformation_arr` (the array containing the roto-translation transformation) as parameters.

After exiting the loop, the `ceres::Solve` function was executed to obtain the transformation.

```
// For each point....
for( int i = 0; i < num_points; i++ )
{
  ceres::CostFunction* cost_function = PointDistance::Create(source_clone.points_[source_indices[i]],
                                                             target_.points_[target_indices[i]]);
  problem.AddResidualBlock(cost_function,
                           nullptr /* squared loss */,
                           transformation_arr.data());
}

ceres::Solve(options, &problem, &summary);
```

Figure 2: Ceres process to use LM to get the estimated transformation

Once Ceres solved the problem, the `transformation_arr` vector was accessed to compute the 4D roto-translation matrix transformation.
The rotation matrix **R** was obtained from the first three angle-axis cells using the `Eigen::AngleAxisd` method.
The translation vector was instead directly extracted from the last three cells.

Finally, the rotation matrix and translation vector were concatenated into a 4D transformation matrix, which was then returned.

Task 5 – `execute_icp_registration`

Task 5 was the most important one because it was the function from which all the other functions were called.
A primary for loop iterates up to a maximum of `max_iteration` (100) times to execute the following steps:

- **Find Closest Points:** Launch `find_closest_point` to get coupled points (the source and target indices) and current RMSE.
- **Check Convergence:** If the absolute difference between previous and current RMSE is less than `relative_rmse` (which is equal to 1e-6 in this case), return because the execution converged.
- **Update RMSE:** Update the `prev_rmse` to `current_rmse`
- **Perform ICP Registration:** Launch `get_lm_icp_registration` if mode == "lm" or launch `get_svd_icp_registration` if mode == "svd"
- **Save Transformation:** Save the returned transformation into `new_transformation`
- **Update transformation_ and source cloud:** Update the `transformation_` 4D matrix and transform `source_for_icp_` using `new_transformation` to avoid recalculating the transformed source cloud each time.

If after `max_iteration` the algorithm hasn't converged, a "Diverged: MAX_ITERATION surpassed." message is sent.

2) Encountered problems

During the assignment, no major problems were encountered.

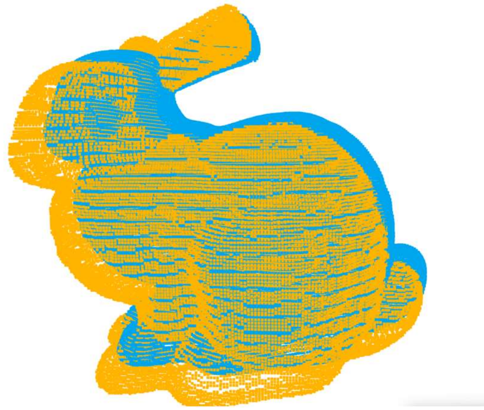
3) Quantitative results

	SVD RMSE	LM RMSE
BUNNY	0.00401621	0.00341366
DRAGON	0.00568867	0.00564134
VASE	0.0162243	0.0162217

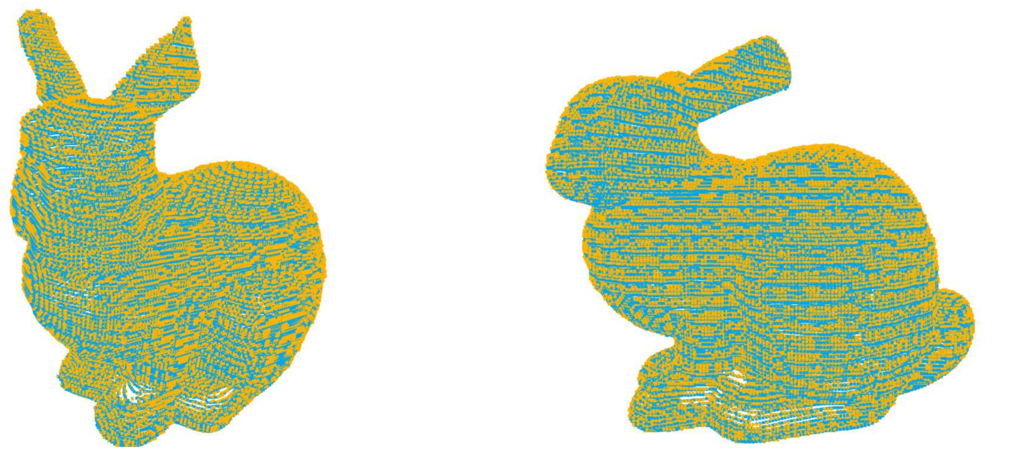
	SVD ITERATIONS	LM ITERATIONS
BUNNY	23	21
DRAGON	13	19
VASE	25	29

4) Qualitative results

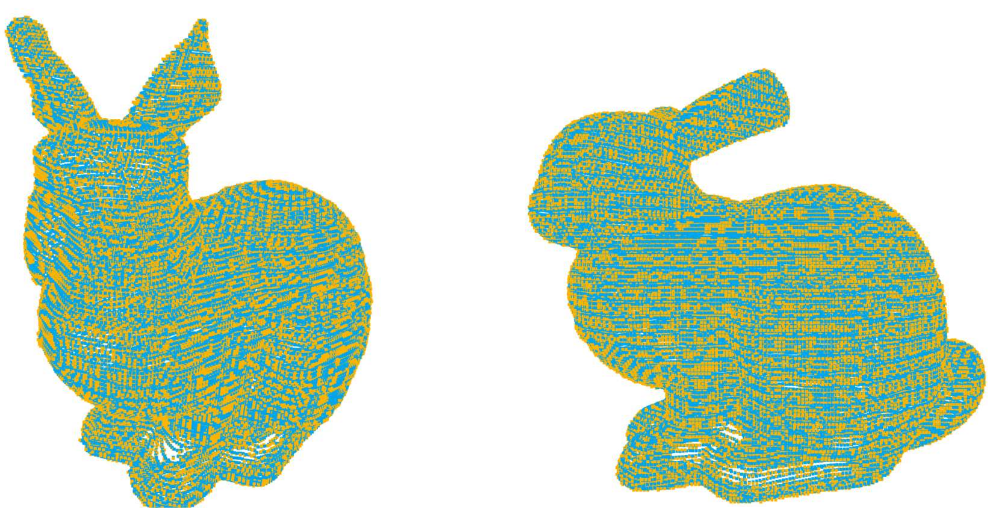
4.1) Bunny



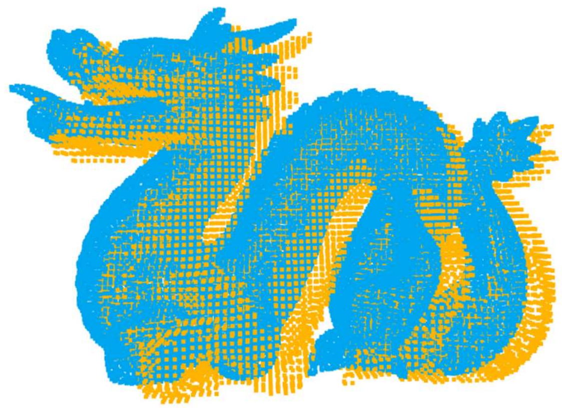
4.1.1) Bunny SVD → 0.00401621 at iteration 23



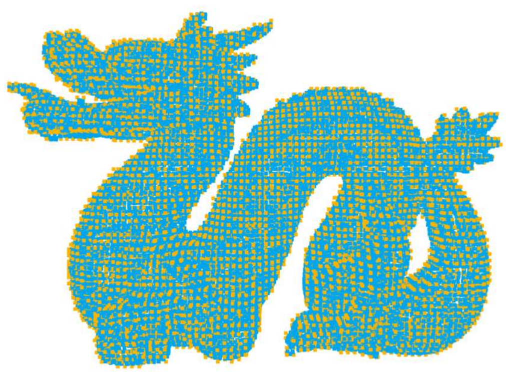
4.1.2) Bunny LM → 0.0341366 at iteration 21



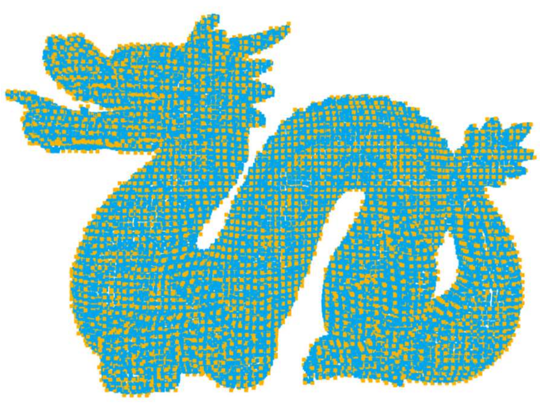
4.2) Dragon



4.2.1) Dragon SVD → 0.00568867 at iteration 13

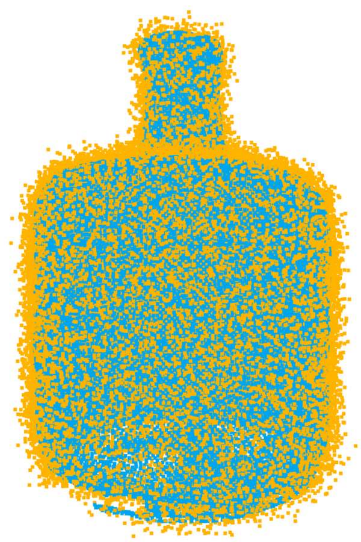


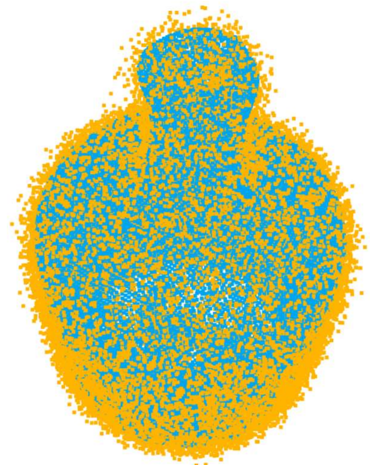
4.2.2) Dragon LM → 0.00564134 at iteration 19



4.3) Vase

4.3.1) Vase SVD → 0.0162243 at iteration 25





4.3.2) Vase LM → 0.0162217 at iteration 29