COMP0119 Report

Poisson Surface Reconstruction

1. Introduction

Surface reconstruction is reconstructing a 3D surface from a set of unorganized sampled points cloud. The primary goal is to produce a continuous and smooth surface that could represent the shape of the object. There are already a number of academic methods for reconstruction, such as RFS, Poisson Surface reconstruction, MLS and naïve reconstruction.

This report is inspired by a paper entitled 'Poisson surface reconstruction' and focuses on understanding and reproducing the methods in his paper, with some additional comparisons and summaries. We will firstly implement an adaptive 'Poisson Surface Reconstruction', a traditional and classic method in reconstruction field. Secondly a comparison between different reconstruction methods will be presented. We will also investigate the latest reconstruction methods, and in particular the impact of deep learning on them in the end.

2. Poisson surface reconstruction

2.1 Core theories

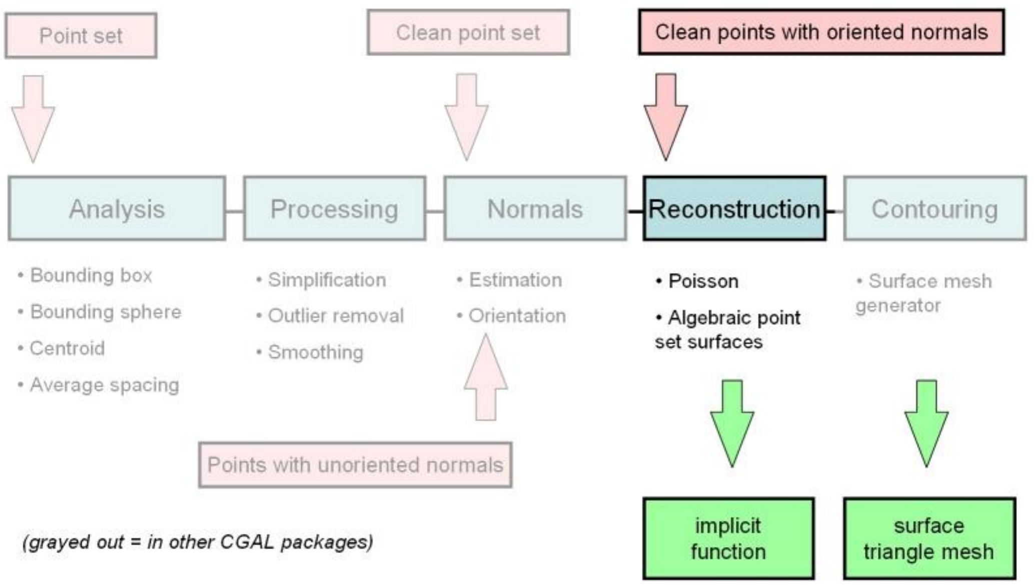


Figure 1: Common Surface Reconstruction Pipeline [1]

Typical surface reconstruction processing has the above pipeline in Figure 1, where point set is firstly analyzed and processed to get boundary and centroid information and is applied subsampling and outlier removal to get a smoothed rudimentary point set. Then, an estimation on the normal of the surface will be done to construct an oriented vector field for reconstruction session. When it has a set of clean points with oriented normal, Prerequisites for using Poisson Surface Reconstruction is satisfied. In the end, mesh of the surface is generated basing on the output of 'Reconstruction' .

The core session of the process is the 'Reconstruction'. The intuitive explanation of the session is finding the best indicator function of the surface that can generate the oriented vector which best match to the normals we obtained in previous session. In mathematics we have:

$$X = argmin_X(||\nabla X - V||)$$

Where X is the indicator function, V is the oriented vectors (could be seen as a variational form of the Normals). ∇ is the gradient operator.

Note: for the module of reconstruction, the input are points and corresponding normals.

However, the author used Poisson system to find the indicator function which in mathematics:

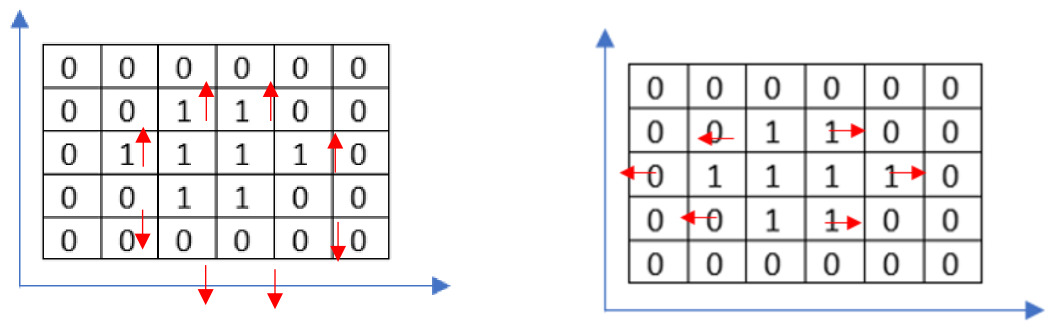
$$X = argmin_X(||\Delta X - \nabla V||)$$

We will discuss it in session 5.

2.2 Procedures or theories that are adapted by the paper

In this part, we will illustrate some assumptions of the paper and justify them in an intuitive way to show our understanding.

Assert: The gradient of the indicator function is equivalent to oriented vector.



(a) (b)

Figure 2: Illustration for the relationship between indicator function and oriented vector

To intuitively explain the assumption, we would simplify the problem to a 2D uniformed image. The grid in Figure 2 represents a 6x5 pixel black and white image. Figure 2a applies a gradient operator along y-axis to the pixels while Figure 2b applies a gradient operation along x-axis with 1 as the interval. It is obvious that the arrows of the vector created by gradient operator can roughly indicate the shape of the object in the center of the image.

Despite the fact that the x-oriented vector cannot match the y-oriented vector in the map precisely (having an offset of 1 pixel), we still can adopt this assumption as in practice we are using sparse matrix to construct operator that will split a vector in 3D into three separate sparse matrix and stack them together. The influence of the offset will be eliminated.

Use Octree to map each point into a local cube

Octree describes the data structure which divides the entire 3D space into 8 equal-sized sub cubes. And each sub-cube can be divided into another smaller 8 cubes and so on. The division operation will stop when a certain level of depth is reached or when no point is inside a sub-cube. The reason on why It is entitled as 'tree' is that the information of each cube is only stored in the parent node of them. This means each cube has and only has one parent node and 8 children nodes, which can be represented as following graph (figure 3).

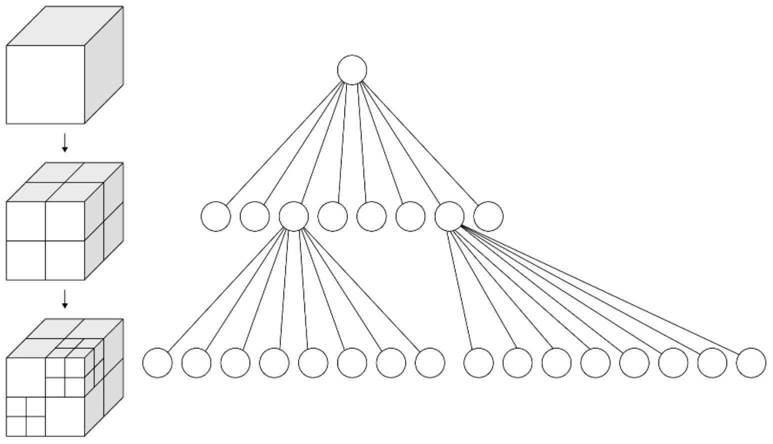


Figure 3: illustration of octree

Similarly, quadtree has nearly the same definition as octree but it represents the image (Shown in figure 4)

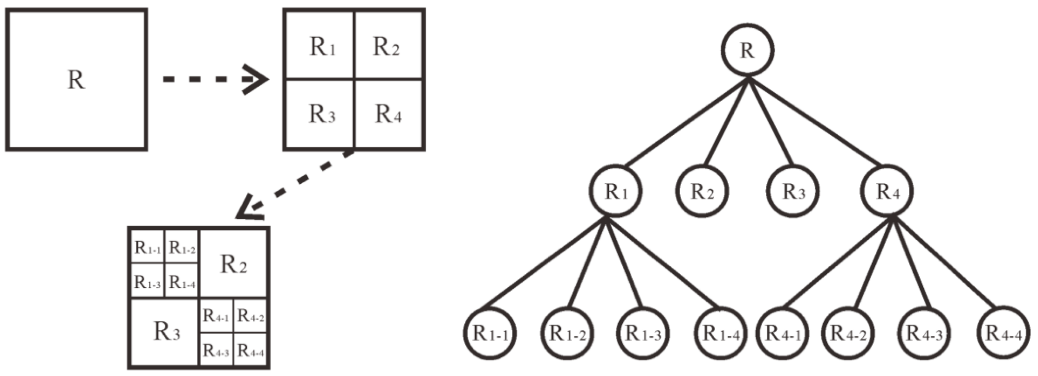


Figure 4: illustration of quadtree

Smooth the surface with trilinear interpolation.

Trilinear interpolation is used to estimate the value of a point within a 3D grid, taking into account the surrounding points in each of the three dimensions. This results in a smoother, more continuous representation of the data and helps to reduce artefacts in the final objects.

Initially, trilinear interpolation is to calculate a certain point within a cube with the knowledge of the 8 neighbors' values. However, in the paper, the author inverted the process as he used it to distribute the normals of the samples to their 8 closest neighbors.

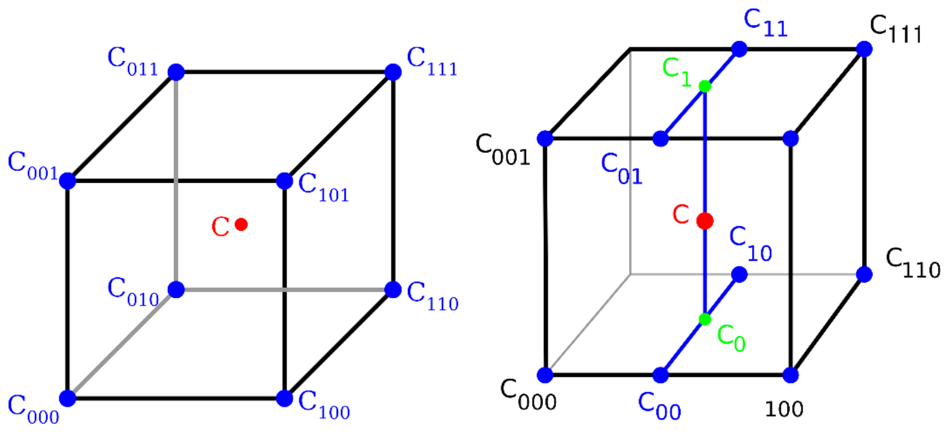


Figure 5: Illustration of trilinear interpolation

Intuitively, suppose we have divided an object into a number of organized cubes. Each point has their location inside a certain cube. Trilinear interpolation is to calculate the weights of each node and project the point's value to each node based on the weights. The size of the weight is inversely proportional to the distance of the sample point from the node. Points in figure 5 with sublabel 0,1,01,11,10,00 are auxiliary points used to calculate weights.

3. Implementation

3.1 Alternative grid segmentation methods

```
# Part 0:# for simplicity, rename the variables
P = self.P
N = self.N
nx = self.nx
ny = self.ny
nz = self.nz
padding = self.padding

# Part 1 : get the information of the point cloud and pad the point cloud

# bounding box size of the point cloud
bbox_size = np.max(P, 0) - np.min(P, 0)
# grid spacing along x, y, and z axis ZJA: divide the box into 64x64x64 small boxes
hx = bbox_size[0] / nx
hy = bbox_size[1] / ny
hz = bbox_size[2] / nz

# the world coordinates of the bottom left front corner of the volume
# at the bottom_left_front_corner
bottom_left_front_corner = np.min(P, 0) - padding * np.array([hx, hy, hz])
# grid numbers along x, y, and z axis, ZJA: I guess the program will pad in both side of one axis
nx += 2 * padding
ny += 2 * padding
nz += 2 * padding
```

Figure 6: code for indexing process and grid segmentation

Octree is efficient in special querying as it partitions 3D space into smaller regions, allowing for faster search operations. However, octrees can be complex to implement and can require significant effort to optimize for specific use cases. This can make them less suitable for applications where simpler data structures are sufficient. Since we don't need to query the position of specific points, but can process them in batches using a sparse matrix approach, using octrees would make the code more complex and difficult to control. We have decided to use a brute-force method, simply dividing the space into (nx, ny, nz) shaped cells. The cells without points may take up some memory but have almost no effect on the computation. For cases where memory is sufficient, this alternative method is still good.

3.2 explicit gradient operator by sparse matrix

To enhance the transparency of the computing process for better analysis and troubleshooting, we have opted not to rely on various third-party integrated functions like scipy.interplate and np.gradient, and instead, we have chosen to have customed control over the program. Furthermore, instead of utilizing octree, we have opted for a simpler alternative approach.

We are going to use the sparse matrix that is commonly used in imaging processing, where an image with shape of n by m will be flattened at the first place. Then an operator with shape of (n*m, n*m) will be constructed for linear operation. Usually, the diagonal of the operation matrix is filled with 1. Each row of the operation matrix represents the formula:

$$a_1 * pixel_1 + a_2 * pixel_2 + \cdots a_i * pixel_i + \cdots a_n * pixel_n = newPixel$$

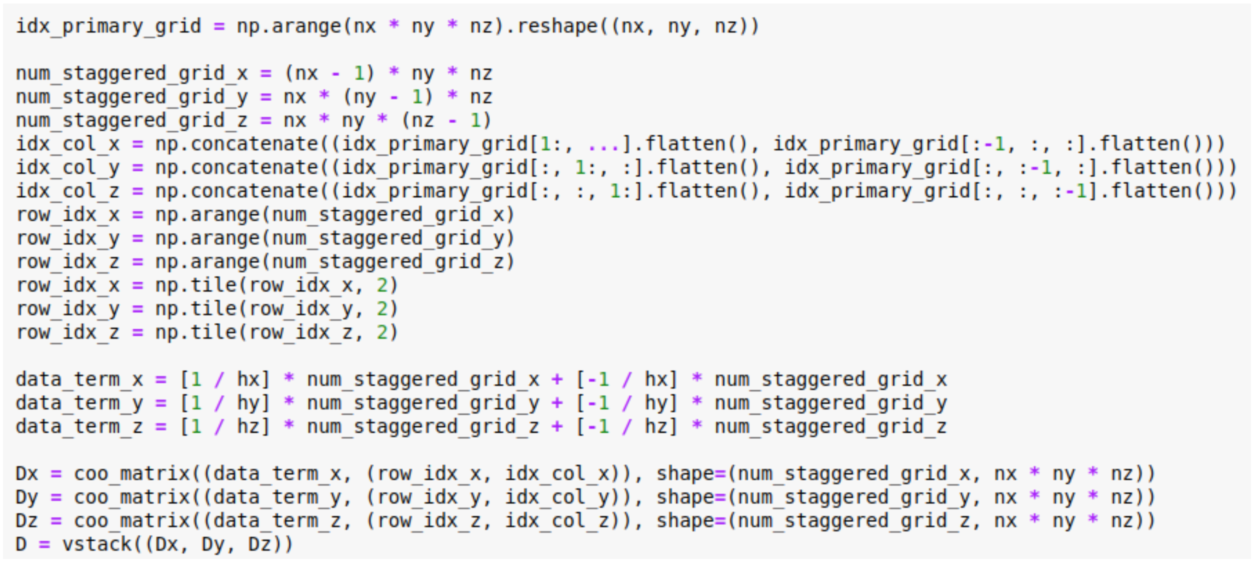
Where pixels are the data themselves and coefficients a are the elements of that row in operation matrix.

Similarly, to find a X (indicator) that satisfied the following formula, the indicator will be flattened.

$$X = argmin_X(||\Delta X - \nabla V||)$$

However, we are going to compare the indicator and the normals in divergence space where we are going to compute the gradient and gradient is a vector quantity. The output of the gradient operator hence should contain the information of all vectors. To achieve this, the operator will be in shape of (3*n*m,n*m).

The code implementation of constructing 3D gradients is shown below.



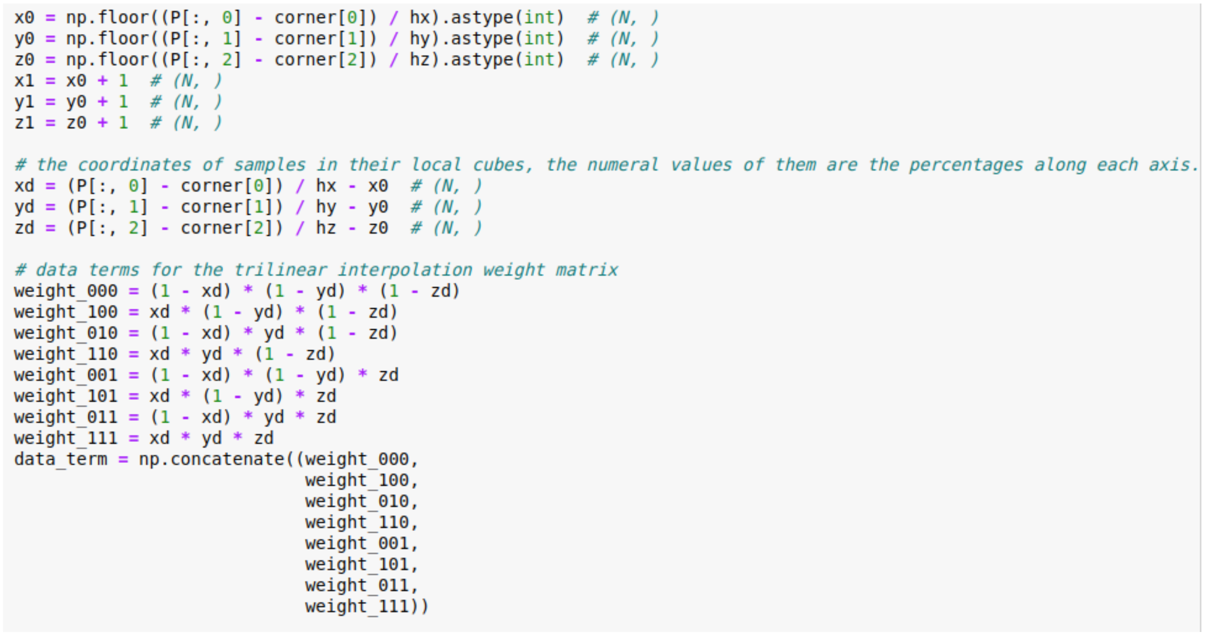
```
idx_primary_grid = np.arange(nx * ny * nz).reshape((nx, ny, nz))

num_staggered_grid_x = (nx - 1) * ny * nz
num_staggered_grid_y = nx * (ny - 1) * nz
num_staggered_grid_z = nx * ny * (nz - 1)
idx_col_x = np.concatenate((idx_primary_grid[1:, ...].flatten(), idx_primary_grid[:-1, :, :].flatten()))
idx_col_y = np.concatenate((idx_primary_grid[:, 1:, :].flatten(), idx_primary_grid[:, :-1, :].flatten()))
idx_col_z = np.concatenate((idx_primary_grid[:, :, 1:].flatten(), idx_primary_grid[:, :, :-1].flatten()))
row_idx_x = np.arange(num_staggered_grid_x)
row_idx_y = np.arange(num_staggered_grid_y)
row_idx_z = np.arange(num_staggered_grid_z)
row_idx_x = np.tile(row_idx_x, 2)
row_idx_y = np.tile(row_idx_y, 2)
row_idx_z = np.tile(row_idx_z, 2)

data_term_x = [1 / hx] * num_staggered_grid_x + [-1 / hx] * num_staggered_grid_x
data_term_y = [1 / hy] * num_staggered_grid_y + [-1 / hy] * num_staggered_grid_y
data_term_z = [1 / hz] * num_staggered_grid_z + [-1 / hz] * num_staggered_grid_z

Dx = coo_matrix((data_term_x, (row_idx_x, idx_col_x)), shape=(num_staggered_grid_x, nx * ny * nz))
Dy = coo_matrix((data_term_y, (row_idx_y, idx_col_y)), shape=(num_staggered_grid_y, nx * ny * nz))
Dz = coo_matrix((data_term_z, (row_idx_z, idx_col_z)), shape=(num_staggered_grid_z, nx * ny * nz))
D = vstack((Dx, Dy, Dz))
```

Figure 7: The code for explicit gradient operator

In the case of a sparse matrix, the gradient is calculated by subtracting the value of the current pixel from the value of the next pixel in the same direction, and then dividing the result by the interval between them. For instance, the sparse matrix, denoted as x, has elements located at specific coordinates in a matrix. The gradient of a pixel at coordinate (i, j) is found by subtracting x[i,j+1] from x[i,j], and then dividing by the interval between them. This interval is represented by h, and the coefficients of x[i,j] and x[i,j+1] in the gradient calculation are 1/h and -1/h, respectively.

3.2 trilinear interpolation

Step 1: get the weights of each point in all cubes.

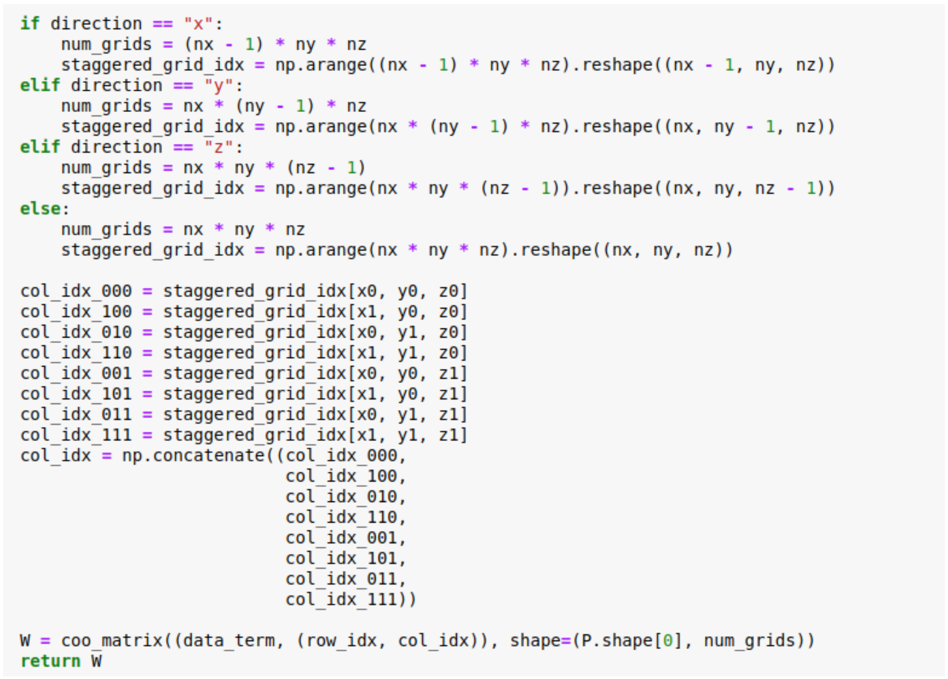


```
x0 = np.floor((P[:, 0] - corner[0]) / hx).astype(int)  # (N, )
y0 = np.floor((P[:, 1] - corner[1]) / hy).astype(int)  # (N, )
z0 = np.floor((P[:, 2] - corner[2]) / hz).astype(int)  # (N, )
x1 = x0 + 1  # (N, )
y1 = y0 + 1  # (N, )
z1 = z0 + 1  # (N, )

# the coordinates of samples in their local cubes, the numeral values of them are the percentages along each axis.
xd = (P[:, 0] - corner[0]) / hx - x0  # (N, )
yd = (P[:, 1] - corner[1]) / hy - y0  # (N, )
zd = (P[:, 2] - corner[2]) / hz - z0  # (N, )

# data terms for the trilinear interpolation weight matrix
weight_000 = (1 - xd) * (1 - yd) * (1 - zd)
weight_100 = xd * (1 - yd) * (1 - zd)
weight_010 = (1 - xd) * yd * (1 - zd)
weight_110 = xd * yd * (1 - zd)
weight_001 = (1 - xd) * (1 - yd) * zd
weight_101 = xd * (1 - yd) * zd
weight_011 = (1 - xd) * yd * zd
weight_111 = xd * yd * zd
data_term = np.concatenate((weight_000,
                            weight_100,
                            weight_010,
                            weight_110,
                            weight_001,
                            weight_101,
                            weight_011,
                            weight_111))
```

Figure 8: The code for trilinear interpolation

The main concept of this trilinear interpolation is to distribute the normal of each point to the grid, following a rule we discussed above. The implementation of it is shown in figure b.

Step 2: According to the direction and calculate the interpolation in each direction.

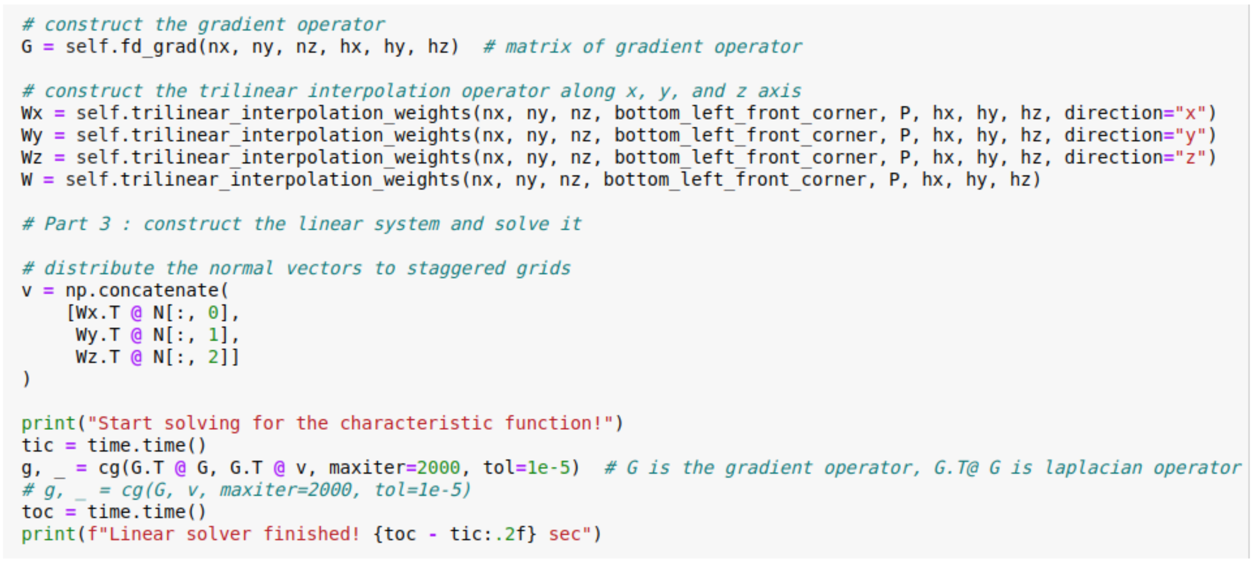


```
if direction == "x":
    num_grids = (nx - 1) * ny * nz
    staggered_grid_idx = np.arange((nx - 1) * ny * nz).reshape((nx - 1, ny, nz))
elif direction == "y":
    num_grids = nx * (ny - 1) * nz
    staggered_grid_idx = np.arange(nx * (ny - 1) * nz).reshape((nx, ny - 1, nz))
elif direction == "z":
    num_grids = nx * ny * (nz - 1)
    staggered_grid_idx = np.arange(nx * ny * (nz - 1)).reshape((nx, ny, nz - 1))
else:
    num_grids = nx * ny * nz
    staggered_grid_idx = np.arange(nx * ny * nz).reshape((nx, ny, nz))

col_idx_000 = staggered_grid_idx[x0, y0, z0]
col_idx_100 = staggered_grid_idx[x1, y0, z0]
col_idx_010 = staggered_grid_idx[x0, y1, z0]
col_idx_110 = staggered_grid_idx[x1, y1, z0]
col_idx_001 = staggered_grid_idx[x0, y0, z1]
col_idx_101 = staggered_grid_idx[x1, y0, z1]
col_idx_011 = staggered_grid_idx[x0, y1, z1]
col_idx_111 = staggered_grid_idx[x1, y1, z1]
col_idx = np.concatenate((col_idx_000,
                          col_idx_100,
                          col_idx_010,
                          col_idx_110,
                          col_idx_001,
                          col_idx_101,
                          col_idx_011,
                          col_idx_111))

W = coo_matrix((data_term, (row_idx, col_idx)), shape=(P.shape[0], num_grids))
return W
```

Figure 9: code for computing the weight of trilinear interpolation

This operator is applied to the normal, which are vectors. Therefore, each direction has its own trilinear interpolation. The implementation is shown in figure 9.

3.3 Compute



```python
# construct the gradient operator
G = self.fd_grad(nx, ny, nz, hx, hy, hz)  # matrix of gradient operator

# construct the trilinear interpolation operator along x, y, and z axis
Wx = self.trilinear_interpolation_weights(nx, ny, nz, bottom_left_front_corner, P, hx, hy, hz, direction="x")
Wy = self.trilinear_interpolation_weights(nx, ny, nz, bottom_left_front_corner, P, hx, hy, hz, direction="y")
Wz = self.trilinear_interpolation_weights(nx, ny, nz, bottom_left_front_corner, P, hx, hy, hz, direction="z")
W = self.trilinear_interpolation_weights(nx, ny, nz, bottom_left_front_corner, P, hx, hy, hz)

# Part 3 : construct the linear system and solve it

# distribute the normal vectors to staggered grids
v = np.concatenate(
    [Wx.T @ N[:, 0],
     Wy.T @ N[:, 1],
     Wz.T @ N[:, 2]]
)

print("Start solving for the characteristic function!")
tic = time.time()
g, _ = cg(G.T @ G, G.T @ v, maxiter=2000, tol=1e-5)  # G is the gradient operator, G.T@ G is laplacian operator
# g, _ = cg(G, v, maxiter=2000, tol=1e-5)
toc = time.time()
print(f"Linear solver finished! {toc - tic:.2f} sec")
```

Figure 10: code for main logic

Initially, the gradient operator is created by taking into account the number of cubes and the intervals in each direction. This operator can be utilized for both indicator and vector fields as they both share the same cube matrix that is defined beforehand. Following this, the interpolation operator is constructed and applied to the normals in order to obtain vector fields. Finally, a linear solver is utilized to compute the desired indicator.

4. Comparison with other approaches

In the approach of "Poisson Surface Reconstruction," the author decided to use the Laplacian to determine the most suitable indicator function. Although they mentioned several advantages of the Poisson system, such as increased resilience to noise, we still lack a direct and intuitive understanding of Poisson surface reconstruction's performance. Therefore, our goal is to compare Poisson surface reconstruction with other commonly used methods.

There are many different academic techniques for reconstructing 3D object surfaces, in addition to the Poisson Surface Reconstruction. In this section, we implemented three alternative reconstructions for comparison, including Naive, Moving Least Squares (MLS), and Radial Basis Function (RBF) reconstruction. We applied these four reconstruction algorithms to 3D objects with small point cloud samples, large point cloud samples, and noisy point cloud samples to evaluate the surface reconstruction outcomes among them.

Initially, a small set of point cloud data was reconstructed using a camel as the test subject. The original camel data contained 1789 vertices. Later, a large set of point cloud data was reconstructed using a dragon as the experimental object, which had 118044 vertices.

4.1 low data comparison

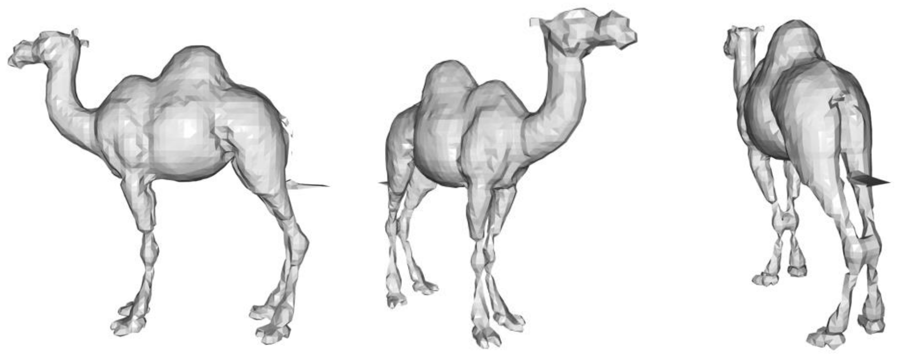


Figure 11: reconstruction from naïve reconstruction

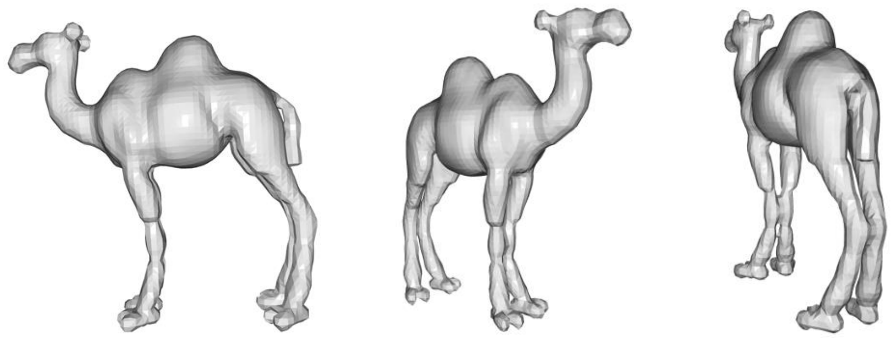


Figure 12. MLS Reconstruction for camel.

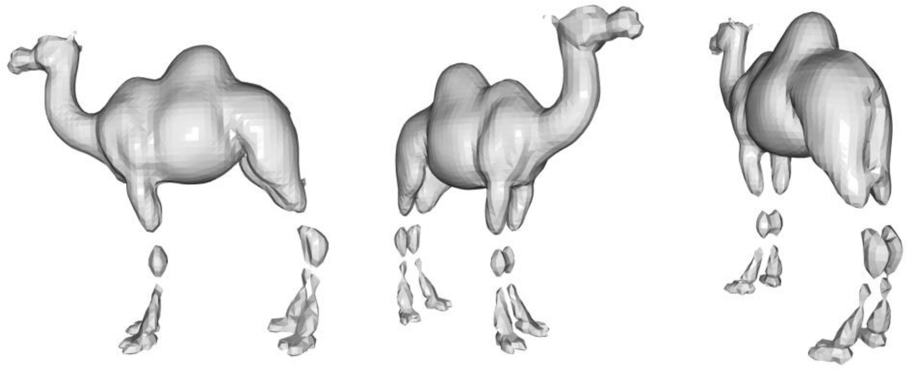


Figure 13. RBF Reconstruction for camel.

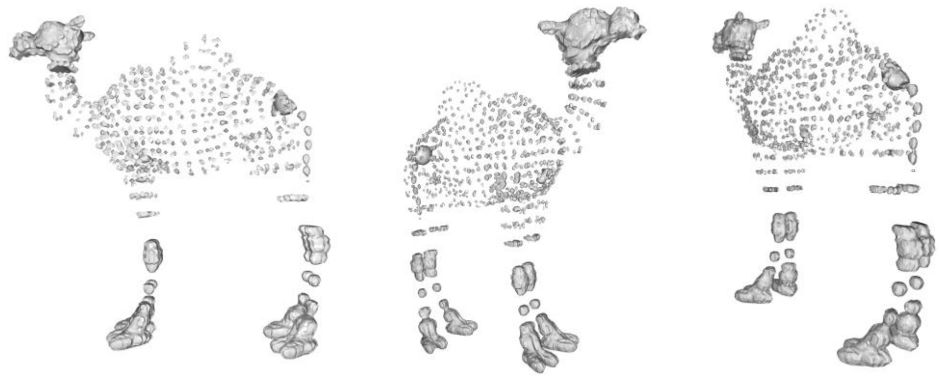


Figure 14. Poisson Reconstruction for camel.

The above reconstruction results demonstrated the performance of three approaches in low data situation. We can find that both the Naive and MLS methods were able to produce surfaces with varying degrees of accuracy and completeness. Naive reconstruction generated a relatively rough surface, but with a high level of completeness. In contrast, the MLS method produced a smoother surface, but there was a risk of oversmoothing the data. This approach abandoned a lot of information from point clouds.

On the other hand, the RBF method produced surfaces that were both smoother and more accurate, closely resembling the point cloud data. However, the reconstruction of the legs of the camel was not successful, possibly due to the nonuniform distribution of the points of the legs, which has less cloud points at that place and led to a discontinuity in the RBF reconstruction area. While RBF is generally considered a good method for surface reconstruction, this limitation indicates that it may not always be the best choice for certain datasets.

On the other hand, the results of the Poisson surface reconstruction method for the camel point cloud dataset were highly unsatisfactory, with minimal surface reconstruction achieved. The reason for this could be attributed to the fact that the Poisson method relies heavily on the point clouds, specifically the normals of the points. The Poisson method distributes the normals to local grids to estimate the indicator function which is the result of linear solver. However, when the point cloud is sparse, many grids may not contain any sample points, leading to a lack of information to determine the boundary of the model. Another possible explanation for the poor performance of the Poisson method could be the roughness of the vertex normals at the beginning, which may have caused the indicator function to indicate the wrong boundary.

Poisson surface reconstruction is known to be more resilient to noise than other methods, but it requires a relatively dense set of points to produce accurate results. Therefore, while it may be effective for larger datasets, it may not be suitable for smaller point clouds.

Overall, the results suggest that the choice of surface reconstruction method depends on various factors, such as the size and density of the point cloud data, and the desired level of accuracy and completeness. Understanding the strengths and limitations of each method is crucial to selecting the most appropriate method for a particular dataset.

4.2 massive data situation

For this experiment, the dragon was used as the test subject, which contains a very large data set of 118044 vertices. The point cloud of this object is highly dense, preserving more detailed mesh features.

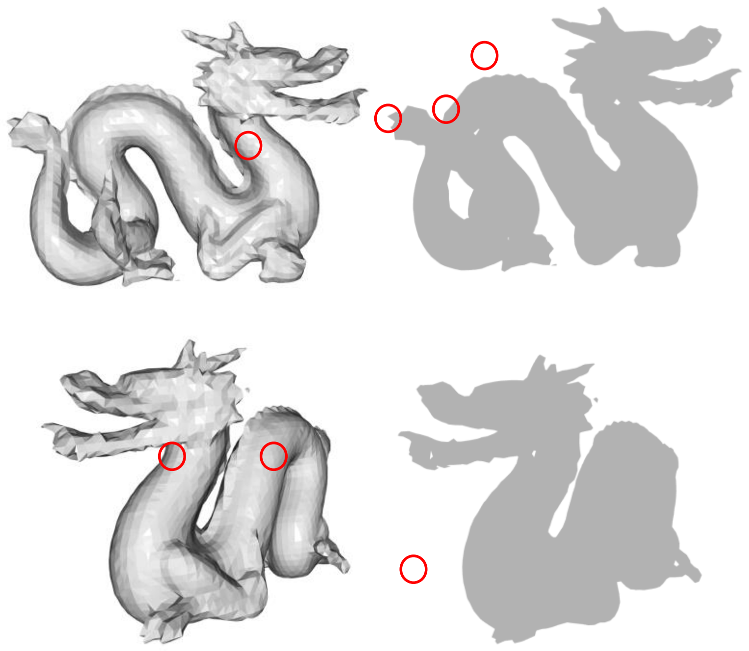
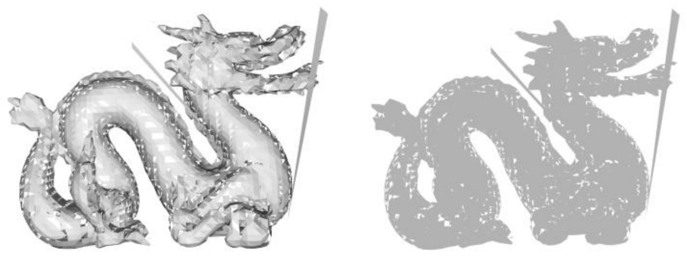


Figure 15. Naïve Reconstruction for the dragon.

The two left meshes displayed in Figure 15 show the rendered Naive reconstructed meshes in different poses, while the two right meshes show the corresponding poses of the non-rendered Naive reconstructed meshes. These non-rendered meshes are used to assess the mesh's integrity after reconstruction. Although the Naive reconstruction algorithm can complete the surface reconstruction of large point clouds quickly with minimal computing resources, the dragon, which is in a huge dataset, is reconstructed with a relatively rough surface. The dragon's surface is made up of many triangular meshes, and many of the dragon's features have not been reconstructed. The non-rendered meshes display many unreconstructed holes on the surface, as indicated by the red circle.



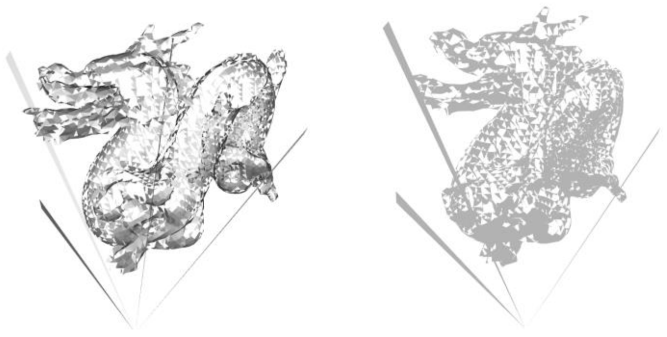


Figure 16. MLS Reconstruction for the dragon.

The outcome of the Moving Least Squares (MLS) reconstruction mentioned above is extremely poor. The dragon's surface is nearly broken up, and there are several gaps and holes, including four artefacts that should not be there. While MLS reconstruction is more computationally intensive than Naive reconstruction, it may not produce satisfactory outcomes for large point cloud samples.

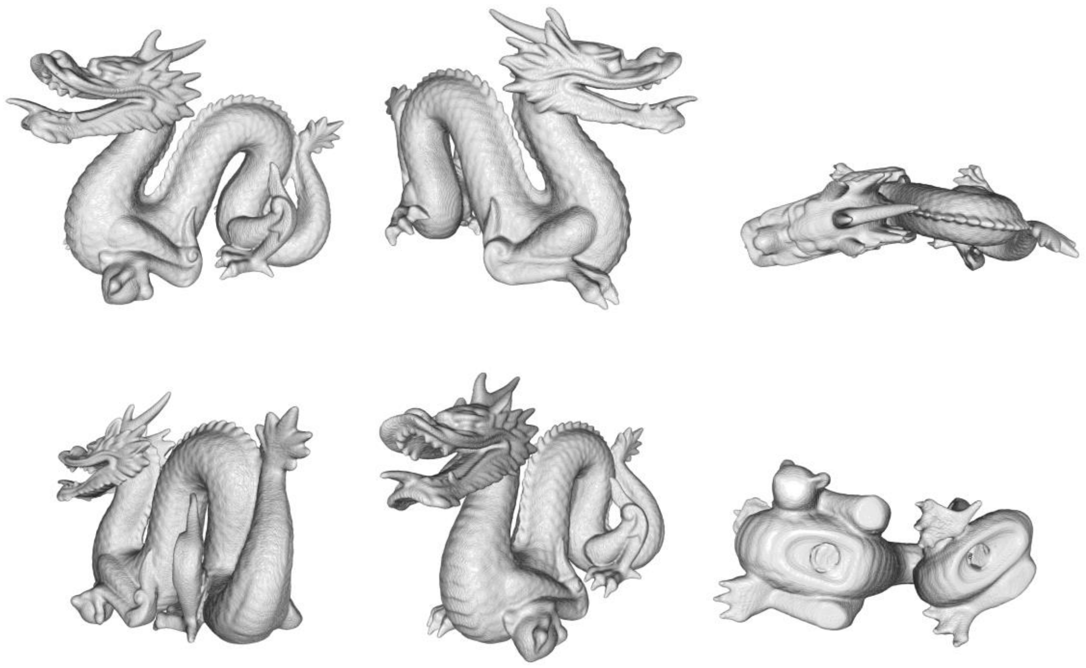


Figure 17. Poisson Surface Reconstruction for dragon.

When dealing with large point cloud data, the Poisson Surface Reconstruction method is able to generate high-quality surfaces that are both smooth and accurate. In comparison to Naive reconstruction, Poisson reconstruction is capable of handling complex surface geometry and achieving precise reconstruction. Additionally, Poisson reconstruction is able to produce surfaces that are watertight and do not contain any holes, ensuring the integrity of the final result.

4.3 noisy data situation

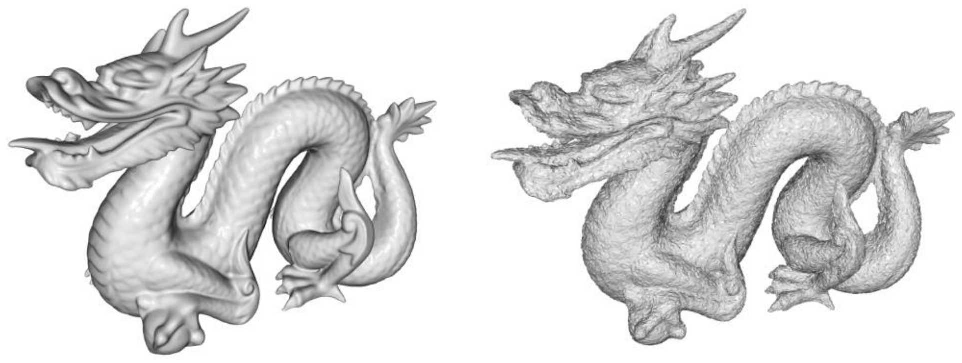


Figure 18. Left: Original Dragon. Right: Noised Dragon.

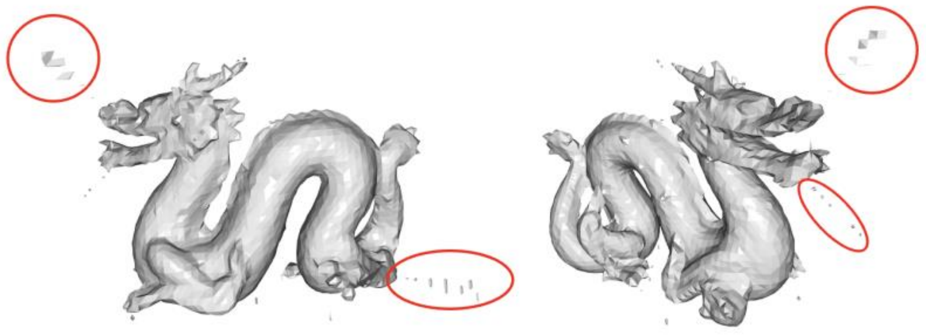


Figure 18: Naïve reconstruction based on noisy samples

When comparing the results in Figure 15 and Figure 18, the presence of noise has an impact on the naive reconstruction method. In Figure 15, there is a lot of noise around the reconstructed model, as can be seen from the red circle. the model's surface has become uneven and irregularly elevated. The presence of noise considerably worsens the outcome of a straightforward reconstruction approach.

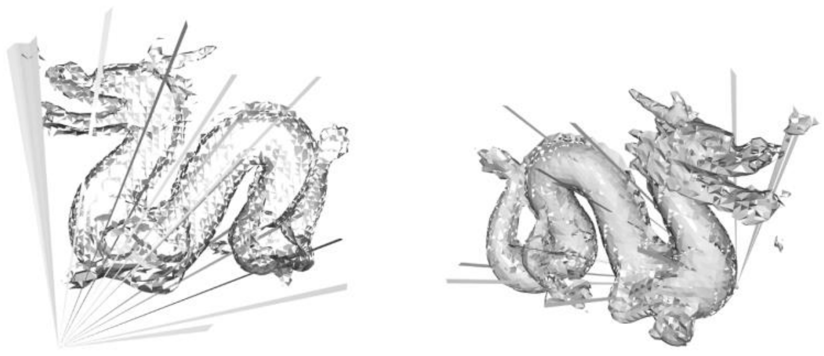


Figure 19: results of MLS reconstruction from noisy samples

The results in Figure 19 indicate that MLS reconstruction is highly sensitive to noise. The presence of noise leads to the presence of more artifacts in the reconstructed model. The surface of the dragon is not fully reconstructed as it would be without the noise.

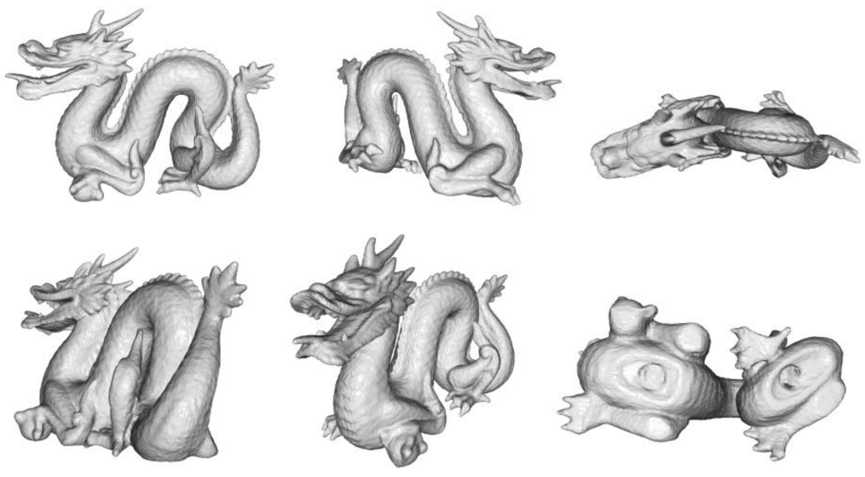


Figure 20: Results of Poisson Reconstruction from noisy samples

In the case of large point cloud data with Gaussian noise, Poisson surface reconstruction proves to be a robust and effective method for producing high-quality surfaces that are both smooth and accurate. Unlike the naive and MLS methods, Poisson surface reconstruction is relatively insensitive to noise in the point cloud data. The resulting surface meshes are continuous and free of holes, and the reconstructed features of the surface are accurately represented. Overall, the Poisson surface reconstruction method appears to be well-suited for dealing with large and complex point cloud data that may contain noise.

5. Latent tech for reconstructions

The development of SLAM has driven researchers to reconstruct 3D objects from 2D images for a better performance of robots sensing their surrounding environment. And the rapid expanding of the computing abilities of GPU has made the existence of Deeper neural networks possible. So, reconstructing 3D objects from 2D images or videos has been well developed in the past decade.

These approaches often involve learning a latent representation of the 3D structure in order to facilitate the reconstruction process. Some popular latent 3D reconstruction techniques include: Autoencoder [2], Generative Adversarial Networks (GAN) [3], DeepSDF [4] and NeRF [5].

5.1 The 3D shape reconstruction based on the Deep Learning

We take Autocoder and GAN as examples.

In the paper "3D ShapeNets: A Deep Representation for Volumetric Shapes," the authors propose a novel deep learning framework for representing and recognizing 3D shapes using a convolutional deep belief network (CDBN). The primary goal is to learn a probabilistic model of 3D shapes that can be used for various tasks, including object recognition, shape completion, and shape generation.

The authors represent 3D shapes as a probability distribution of binary variables on a 3D voxel grid. This representation allows for easy integration with deep learning techniques, as well as a straightforward way to handle partial or noisy observations.

In the paper of Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling, The proposed 3D-GAN consists of a generator network that creates 3D objects from random noise and a discriminator network that evaluates the quality of the generated objects. These two networks are trained together in an adversarial manner, with the goal of producing realistic and diverse 3D shapes.

Additionally, the authors introduce a 3D-VAE-GAN, which combines a 3D Variational Autoencoder (3D-VAE) with the 3D-GAN. The 3D-VAE-GAN is designed to learn a probabilistic latent space of object shapes, allowing for the generation of new shapes by sampling from the latent space.

5.2 Following work of Poisson surface reconstruction.

Michael Kazhdan further developed research on Poisson surface reconstruction, creating the "Poisson Surface Reconstruction with Envelope Constraints" approach [6]. The paper introduces an improved method for reconstructing surfaces from 3D scans, mainly addressing the problem of unobserved surfaces, which can lead to unwanted surface patches when using global optimization techniques.

To solve this issue, the authors adapt the Screened Poisson Reconstruction (SPR) method by incorporating an additional input: a watertight mesh enclosing the space where the reconstructed surface should be. They rasterize the envelope mesh into an adapted octree, perform a flood-fill to identify octree leaf nodes outside the surface, and adjust the finite-element basis to prevent overlapping exterior nodes. This ensures the implicit function vanishes outside the envelope surface.

The approach combines space carving benefits with global method robustness by integrating line-of-sight constraints into a global solver. This results in improved surface reconstruction, avoiding extraneous surfaces and providing a more accurate model.

6. Conclusion

In this report, we first implemented the Poisson Surface Reconstruction (PSR) in Python with as few third-party libraries as possible. Next, to gain a more intuitive understanding of the effectiveness of PSR, we tested it on few points models, massive points models, and noisy massive points models. We also tested MLS, RBF, and naïve reconstruction for comparison. Through this comparison, we confirmed the characteristics mentioned in the literature that PSR indeed has strong adaptability to noise but performs poorly with low data volume. Finally, we briefly reviewed the latest 3D reconstruction methods, especially deep learning for object reconstruction in SLAM. We also found the latest article by the PSR author and summarized it.

Reference

[1] CGAL Project, "CGAL User and Reference Manual: Poisson Surface Reconstruction," CGAL Project, 2023. [Online]. Available: https://doc.cgal.org/latest/Poisson_surface_reconstruction_3/index.html. [Accessed: 10-04-2023].

[2] Z. Wu et al., "3D ShapeNets: A Deep Representation for Volumetric Shapes," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., June 2015, pp. 1912-1920.

[3] J. Wu et al., "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," in Advances in Neural Information Processing Systems 29 (NIPS 2016), 2016, pp. 82-90.

[4] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, "DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., June 2019, pp. 165-174.

[5] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis," in Proc. Eur. Conf. Comput. Vis., 2020, pp. 405-421.

[6] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in Proceedings of the Fourth Eurographics Symposium on Geometry Processing, 2006, pp. 61-70.