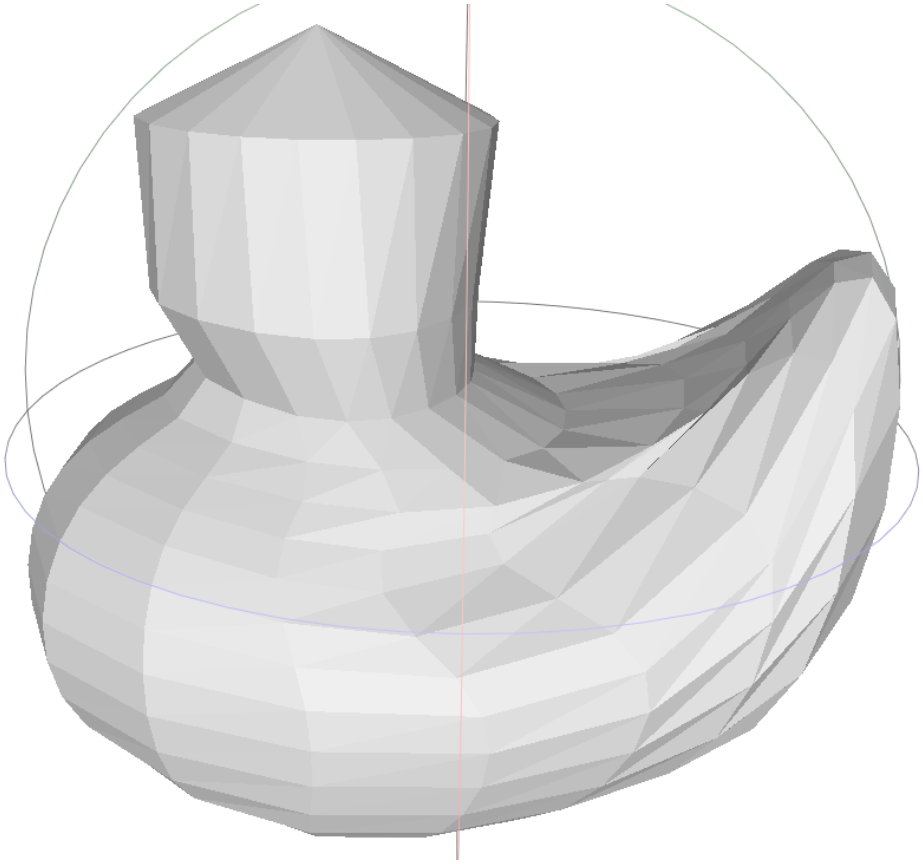
# Unsupervised Shape representations for 3D reconstruction



Master Thesis
6th March 2023 - 6th September 2023

by Shaoyan Li, 20-741-278

Supervisors:
Prof. Dr. Renato Pajarola
Lizeth J. Fuentes Perez

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

University of Zurich[UZH]

VISUALIZATIONANDMULTIMEDIALAB

# Abstract

Non-uniform rational B-Spline surfaces (NURBS surface), a kind of parametric surface, are widely used in 3D modeling. This work explores NURBS surface reconstruction via the NURBS-Diff module. The NURBS-Diff module enables NURBS surfaces differentiable using the PyTorch framework. With supervised parameters, the module reconstructs the NURBS-based point cloud efficiently. This work introduces several pipelines by utilizing the NURBS-Diff module in unsupervised cases. The unsupervised pipelines make use of supersampling methods to obtain unstructured input and propose various metrics for point cloud and surface evaluation. The baseline unsupervised method is adapted from the original supervised pipeline. An extension of the NURBS-Diff module is also presented. The unsupervised pipelines are evaluated against the baseline. The pipelines serve as a stepping stone to further investigation into NURBS surface reconstruction based on unstructured input.

# Contents

# 1 Introduction

## 1.1 Motivation

In recent years, Computer-Aided Design (CAD) [SRN08] software programs are widely used in the field of architecture, mechanical engineering as well as computer science. Artists, designers and researchers more and more rely on this kind of software to create 3D models, which can be stored as models for further use. However, there are situations where it is incapable to access the original models. For instance, there is only a point cloud[1], which is sampled from the original model. However, even if we have the access to the original model, it may be too large to be processed efficiently. Besides, the low resolution of the model may be unsuitable for our need. Therefore, it is necessary to propose some methods to reconstruct the model with fewer data or increase the resolution of the original model.

One common scenario to use CAD software is to create and store precision instrument and component drawings since these drawings require high levels of accuracy. It is necessary to model such instruments since redesigning these machines is so difficult and pricey. However, the aforementioned problems arise, which means the saved files may be too large to be loaded and processed effectively, while it is different to increase the resolution on existing designs. Therefore, to address the problems, parametric surface[2] draws our attention. Since Non-Uniform Rational B-Spline (NURBS) surfaces[3] are widely created in CAD software, their properties are worthy to be explored. NURBS surfaces are a kind of parametric surface, which literally means they are defined by a set of parameters, allowing for generating either smooth surfaces or sharp corners. Additionally, NURBS surfaces can be efficiently stored by their parameters rather than a lengthy list of vertices, edges, and faces.

The focus of this research is to find a feasible approach to reconstruct 3D models from unstructured point clouds using NURBS surface definition. We start from exploring structured input with supervised learning settings, and transition to unordered input with unsupervised learning settings. In the thesis, all study is based on a hypothesis that meshes are only used to generate sampled point clouds, indicating there is no direct usage of meshes. However, converting a point cloud to a NURBS surface is a non-trivial task. The corresponding reasons will be explained in subsequent sections. In short, our primary focus is limited to retrieving a NURBS-based point cloud from a non-NURBS point cloud while mesh reconstruction is a supplementary task.

## 1.2 Outline

In this thesis, several concepts are introduced which are relevant and significant for understanding the following chapters. The related papers are discussed in detail and one of these provides us with a plug-and-play module. Next, the related pipeline described in the original papers is explained. Following that, the advantages and drawbacks of these pipelines are discussed. The main part of this thesis focuses on the approaches of our customized pipelines and their implementation. We extensively discuss the approach details and present the results we have obtained. Afterwards, we evaluate the results with qualitative and quantitative analysis. Finally, we conclude the thesis with a summary and an outlook.

### 1.2.1 Contribution

The thesis aims to find a novel way to reconstruct 3D objects from point clouds, by using NURBS surface algorithm. However, this is just one step to the final goal. In this thesis, we mainly focus on how to convert

[1] https://en.wikipedia.org/wiki/Point_cloud

[2] https://en.wikipedia.org/wiki/Parametric_surface

[3] https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline

a raw point cloud to a NURBS-based point cloud, which is rather challenging since the quality of the point cloud can not be guaranteed. NURBS surfaces are widely used in CAD due to the flexibility and accuracy. Therefore, a surface can easily be obtained by its corresponding parameters. Imagine we could approximate any object or mesh with one or several NURBS surfaces, in this case, the need for training specific neural networks for meshes is eliminated. For instance, we need to process watertight and non-watertight meshes separately by leveraging different models. While NURBS surfaces only focus on the ground truth parameters, the surfaces can process all data uniformly regardless of open or closed surfaces. The thesis explores various aspects, from NURBS objects to general objects, from supervised learning to unsupervised learning, and from specific cases to generalized scenarios. Since not all aspects are covered within the scope of the thesis, the limitations and future work are discussed in the end of this thesis.

The implemented program is written in Python[4] and utilizes several libraries such as Pytorch[5] [PGM+19] for deep learning, PyQt5[6] for visualization. In general, it takes point clouds or parameters as the input and outputs a converted point cloud after the training. The input and output point clouds should be aligned except that the output point cloud is NURBS-based. The code can be used with basic knowledge of NURBS. It has been tested on Windows 10 and is available on github[7].

[4]https://www.python.org/
[5]https://pytorch.org/
[6]https://pypi.org/project/PyQt5/
[7]https://github.com/SyLi9527/NURBSDiff/tree/dev

# 2 Technical background

Several concepts introduced in the following are critical for understanding this thesis and related papers.

## 2.1 Point cloud

Point clouds are widely used in 3D reconstruction since they are easy to obtain. A point cloud can be considered as a cluster of many points while each point in it is in 3D coordinates. In this thesis, point clouds play an important role in terms of computing distance and evaluating alignment. This is because if we want to compare two surfaces, direct comparison does not work. However, if a sufficient number of points are sampled from the surface, comparing the alignment and distribution of these point clouds alone is adequate. The alignment indicates how close two point clouds are, while the distribution reflects the density of the sampling.

### PLY, OFF and OBJ

The PLY file format is the most commonly used format for storing point clouds. The structure of a PLY file is shown in Listing 2.1. The first line is the format of the file. Following that, several lines contain general information regarding points, edges, and faces. Finally, there are the details of the components in order.

**Listing 2.1: PLY file structure**

```
ply
format ascii 1.0
comment VCGLIB generated
element vertex 2509
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 0
property list uchar int vertex_indices
end_header
-74.37138 -248.7037 -550.2237 -0.863588 0.164234 0.411523
120.5308 37.78462 -890.4736 -0.00324953 0.924685 0.276958
3.808455 7.865408 -613.7413 -0.458634 0.852259 -0.187183
...
```

OFF and OBJ files are always used for storing mesh information, but they can save point clouds as well. Besides, NOFF is a variant of OFF file with normals attached to each point. The structure of OFF (NOFF) file is depicted in Listing 2.2 and 2.3, which is very similar to PLY file. For the sake of simplicity, this study mainly copes with OFF files.

**Listing 2.2: OBJ file structure**

```
####
#
# OBJ File Generated by Meshlab
#
####
# Object cat_2500_obj.obj
#
```

```
# Vertices: 2509
# Faces: 0
#
####
vn -0.863588 0.164234 0.411523
v -74.371384 -248.703705 -550.223694
vn -0.003250 0.924685 0.276958
v 120.530800 37.784618 -890.473572
...
```

**Listing 2.3: OFF file structure**

```
NOFF
2509 0 0
-74.37138 -248.7037 -550.2237 -0.863588 0.164234 0.411523
120.5308 37.78462 -890.4736 -0.00324953 0.924685 0.276958
...
```

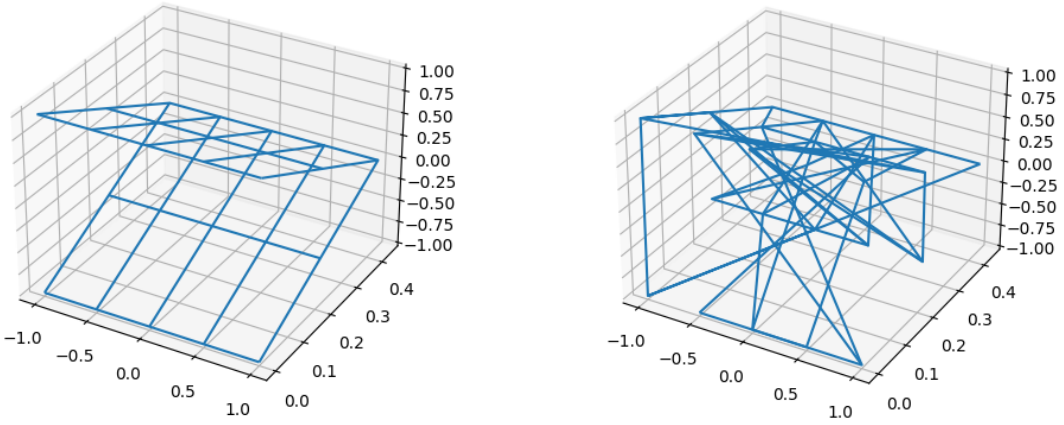
In the second row of this OFF file, There are three separate numbers showing the number of vertices, faces and edges in order. Subsequent rows are the coordinates of vertices and normals. If the object contains any faces or edges, the rows will continue after vertices.

## The order in a point cloud

The order of points within a point cloud is critical for the NURBS reconstruction process.

Typically point clouds can be loaded from file formats like PLY or OFF as mentioned in the previous subsection and stored in a NumPy array [HMvdW+20]. The order of a point is revealed from the position of the point in the array or the original file. Since 3D scenario is concerned in this research, wireframe[1] model is a feasible solution to visualize the structure of point clouds. So by leveraging *plot_wireframe* in package Matplotlib [Hun07], we are able to visualize the topology of any point cloud. This method needs a 2D grid, therefore, the input should be in 3D format correspondingly. For instance, if our input size is (7, 10, 3), it means there is a 2D grid with 7 rows and 10 columns, while each point on the grid is a 3D point. Like other grids, edges of the wireframe model are connected between adjacent points in 2D parametric space. The point $(x, y, z)$, whose coordinate in parametric space is $(u, v)$, should be connected to all points whose corresponding 2D points are the neighbors of $(u, v)$. Therefore, if those neighbors exist within the bounds of the grid, their matching 2D points should be $(u-1, v)$, $(u+1, v)$, $(u, v-1)$ and $(u, v+1)$. According to the connectivity relationship, the points should also be connected in 3D space if they are connected in 2D space.

[1] https://en.wikipedia.org/wiki/Wire-frame_model




(a) Ordered points

(b) Shuffled points

Figure 2.1: Different point cloud orderings.

Figure 2.1(a) demonstrates the order of points in a curved plane. If points are shuffled, the resulting wireframe object will be changed as shown in Figure 2.1(b). In the following sections, point order in NURBS surface will be illustrated in detail.

## 2.2 Poisson disk sampling

Poisson disk sampling[2] [Jon06, Bri07] is a kind of supersampling method. The algorithm allows random sampling of points while ensuring no two points are close to each other. The Poisson disk sampling algorithm is extensively utilized in our work due to its efficiency and speed.

## 2.3 Parametric surface

A parametric surface is a surface in the Euclidean space $R^3$ which is defined by a parametric equation with two parameters $\mathbf{r}$: $D \rightarrow R^3$, where $D$ is a subset of the Euclidean plane[3] $R^2$. By converting 2D points in the parametric space to 3D space, surface parameterization is a process of using parameters to represent a surface.

NURBS surface is a kind of parametric surface, which works on top of the 2D parametric domain $(u, v)$. Typically, 2D space $(u, v)$ can be considered as a 2D grid ranging from 0 to 1 for both $u$ and $v$, with infinite rows and columns. By defining sample resolution along $u$ and $v$, the points on the surface can be evaluated, which means once the parameters of a surface are obtained, the corresponding sampling points can be computed directly from the surface. Therefore parametric surface reconstruction is straightforward because the only key is to obtain related parameters.

[2]https://en.wikipedia.org/wiki/Supersampling#Poisson_disk
[3]https://en.wikipedia.org/wiki/Euclidean_plane

## 2.4 B-spline

A B-spline[4](or basis spline) is a type of spline[5] with well-defined basis functions which are computed using de Boor's algorithm[6]. The shape of a B-spline is determined by its control points, knots, basis functions and degree. NURBS is a special case of B-splines.

## 2.5 NURBS

NURBS stands for non-uniform rational B-spline [WW91, PT97]. NURBS curves are omitted since our main focus is 3D reconstruction. A NURBS surface, which can be considered as the tensor product of two NURBS curves, is defined by the following parameters: control points **P**, weights **W**, knot vectors **U** and **V**, and degrees **p** and **q**. It is important to note the degrees **p** and **q** are always the same. The Weights of NURBS surfaces add more flexibility over the shape in comparison to B-spline surfaces. Since NURBS surfaces are parametric surfaces, therefore, the formula of NURBS surfaces can be written as follows:

$$\mathbf{S}(u,v) = \sum_{i=1}^{n}\sum_{j=1}^{m}\mathbf{P}_{i,j}R_{i,j}(u,v) \tag{2.1}$$

where $n$ and $m$ are the number of control points in $u$ and $v$ direction, $\mathbf{P}_{i,j}$ is the control point at $i$th row and $j$th column while $w_{i,j}$ is the corresponding weight. The function $R_{i,j}(u,v)$ is defined as the following:

$$R_{i,j}(u,v) = \frac{N_{i,p}(u)N_{j,q}(v)w_{i,j}}{\sum_{a=1}^{n}\sum_{b=1}^{m}N_{a,p}(u)N_{b,q}(v)w_{a,b}} \tag{2.2}$$

where $p$ and $q$ are the degrees in $u$ and $v$ direction, $N_{i,p}(u)$ and $N_{j,q}(v)$ are the basis functions in $u$ and $v$ direction. $N_{i,p}(u)$ is defined as:

$$N_{i,p}(u) = \frac{u-u_i}{u_{i+p}-u_i}N_{i,p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}}N_{i+1,p-1}(u) \tag{2.3}$$

$u_i$ is the $i$th knot in the knot vector $U$. We can notice the basis function $N_{i,p}(u)$ is a recursive function, which means we are required to calculate the basis function based on lower degree basis functions. The initial condition is $N_{i,0}(u) = 1$ if $u_i \leq u < u_{i+1}$ and $N_{i,0}(u) = 0$ otherwise. We will discuss the composition of NURBS surfaces in detail in the following subsections.

### Control points

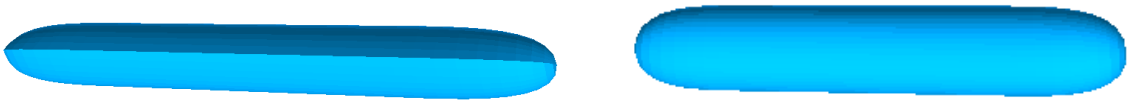
Control points **P** are a set of 3D points. If the number of rows and columns in **P** is defined as $c_u$ and $c_v$ respectively, then the relationship among the number of points in **P**, $c_u$, and $c_v$ can be calculated as Equation 2.4.

$$|\mathbf{P}| = c_u \times c_v \tag{2.4}$$

In 3D space, control points form a control polygon that defines the shape of the NURBS surface. The control polygon is a convex hull of the control points, which indicates all points on the surface should be inside the control polygon. Like a fish net, the control polygon encapsulates the underlying surface. In this way, control points restrict and define the shape of the NURBS surface.

There are two main factors of control points which largely influences the shape of a NURBS surface. the One is the number of control points, while the other is the ordering of control points.

First, the quantity of control points is related to the underlying surface. As illustrated in Figure 2.2, there are two cylinders with different numbers of control points. The cylinder in Subfigure 2.2(b) contains more control points than those in Subfigure 2.2(a). Generally, more control points should lead to better control over the shape.



(a) A cylinder with 7 x 5 control points

(b) A cylinder with 7 x 9 control points

Figure 2.2: Two cylinders with different number of control points

Therefore, increasing the number of control points, like the process of knot insertion, enables the generation of more complex surfaces.

Second, the order of control points also greatly influences the shape. Figure 2.3 compares two cylinders with the same number of control points arranged in different order. Subfigure 2.3(a) shows a normal cylinder defined by $7 \times 9$ control points, while Subfigure 2.3(b) depicts a cylinder with $9 \times 7$ control points, which is the transpose of the control point matrix from previous one. As shown in the figure, this transposed cylinder twists, showing the significance of sequential ordering of control points.

(a) A cylinder with 7 x 9 Control points

(b) A cylinder with 9 x 7 Control points

Figure 2.3: Two cylinders with different ordering of control points

### Weights

Weights vector $W$ is a vector in which each element ranges from 0 to 1. The number of weights is equal to the number of control points. The weights vector solely sets the weights of control points, which refers to the extent how much the control point influences the shape of the surface. In most cases, for the sake of simplicity, the weights vector is set to be a vector of ones, which means all control points have the same influence on their respective controlled batches.

### Degree

The degree of NURBS surface $D$ is a positive integer which determines the degree of basis functions. If the degree is two, then basis functions are quadratic functions. While if the degree is three, then basis functions are cubic functions. As the degree increases, the basis functions become more complex, which enables the surface with finer details and flexibility. Also, the degree of NURBS surfaces is related to another concept named order, as the order is equal to the degree plus one. In most cases, the degree $p$ in $u$ direction and the degree $q$ in $v$ direction are identical, which indicates $D = p = q$ always exists.

### Knot vector

Knot vectors $U$ and $V$ are another key component of NURBS surfaces. From Equations 2.1 and 2.3, knot vectors control the shape of its corresponding NURBS surface, so knot vectors indirectly control the shape of NURBS surface.

Knot vectors are a set of non-decreasing real numbers. One property of knot vectors is the scaling of knot vectors does not change the shape of the surface. In other words, if we scale the knot vectors by a constant, the

[4]https://en.wikipedia.org/wiki/B-spline
[5]https://en.wikipedia.org/wiki/Spline_(mathematics)
[6]https://en.wikipedia.org/wiki/De_Boor's_algorithm

shape of the surface will not change. This is because the ratio between two adjacent knots remains the same. According to Equation 2.3, the coefficients of basis functions are only related to the ratio between two adjacent knots. Therefore, the shape of the surface will not change if we scale the knot vectors by a constant.

Another property of knot vectors is the number of knots in $u$ or $v$ direction is equal to the number of control points in this direction plus the degree plus one. The formula can be shown as follows:

$$\begin{aligned} |U| &= c_u + p + 1 \\ |V| &= c_v + q + 1 \end{aligned} \tag{2.5}$$

This property is also important since it indicates the number of knots is not arbitrary, which is restricted by the degree and number of control points. This is extremely useful in knot insertion or knot removal.

There are some properties of knot vectors which are worth mentioning, such as knot insertion and knot removal. Knot insertion is a process of inserting a knot into current knot vector, which enables the shape to remain the same by adjusting the positions of some control points. Knot removal is a reverse process of knot insertion, which removes a knot from current knot vector. These operations are highly useful as they provide a way to achieve finer or coarser control over the surface.

### Order

The order of a NURBS surface reflects the number of nearby control points which influence points on the surface. The order of a NURBS surface is one more than the degree mathematically. In this way, a third-order NURBS surface is the tensor product of two quadratic NURBS curves while a fourth-order NURBS surface is the tensor product of two cubic curves.

### Basis function

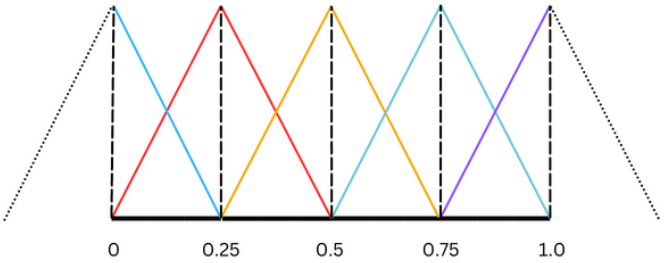



Figure 2.4: The 1-degree basis functions of knot vector [0,0, 0.0, 0.0, 0.25, 0.50, 0.75, 1.0, 1.0, 1.0]

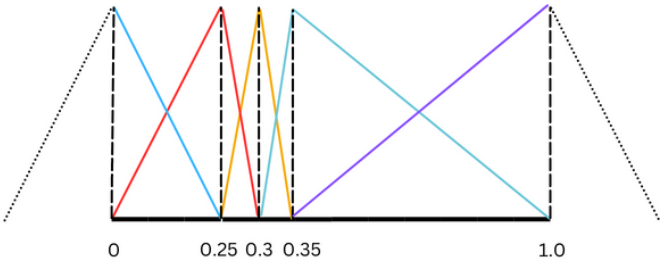



Figure 2.5: The 1-degree basis functions of knot vector [0,0, 0.0, 0.0, 0.25, 0.3, 0.35, 1.0, 1.0, 1.0]

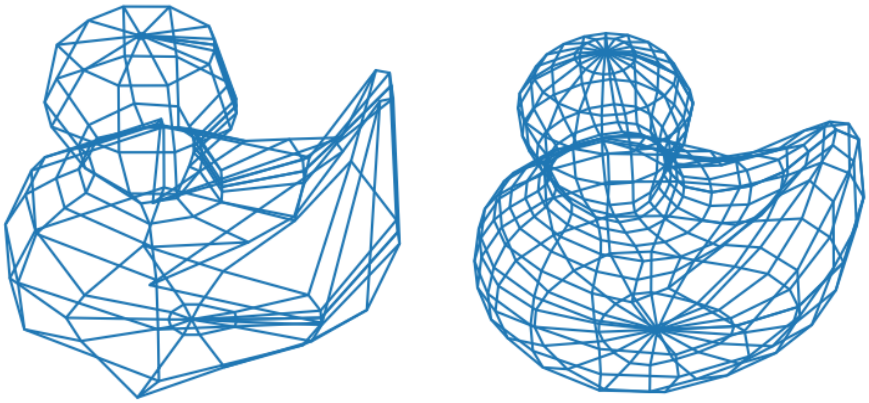
As described in Equation 2.3, the basis functions are defined by the degree and knot vectors. If knot vectors along $u$ and $v$ directions are uniform, then the shapes of basis functions are the same except that the position of each basis function differs. For instance, if the knot vector is set to be [0,0, 0.0, 0.0, 0.25, 0.50, 0.75, 1.0, 1.0, 1.0] and the degree is one, all basis functions are linear functions as shown in Figure 2.4. If two adjacent knots are close in a knot vector, for instance, it is set to be [0, 0, 0.25, 0.3, 0.35, 1.0, 1.0, 1.0], then the distribution of basis

functions is demonstrated as follows in Figure 2.5. By comparing the figures above, we can notice basis functions in the domain of shorter span change rapidly. Even the span can be 0, which indicates knots can be duplicated, This is referred to as knot multiplicity. When a knot has multiplicity, it results in a sharp corner in the surface due to the abrupt change in the basis functions. In most cases, the degree of a NURBS surface is set to three, which generates cubic functions which are easy to compute and flexible.

### Surface evaluation

The process of NURBS surface evaluation is to compute points from a NURBS surface. With specified $u$ and $v$ sampled from knot vectors $U$ and $V$, it is not difficult to evaluate any point from the surface according to Equation 2.1. Therefore, with the help of the formula of NURBS surfaces, a point cloud which is sampled from the ground truth surface can be easily obtained.

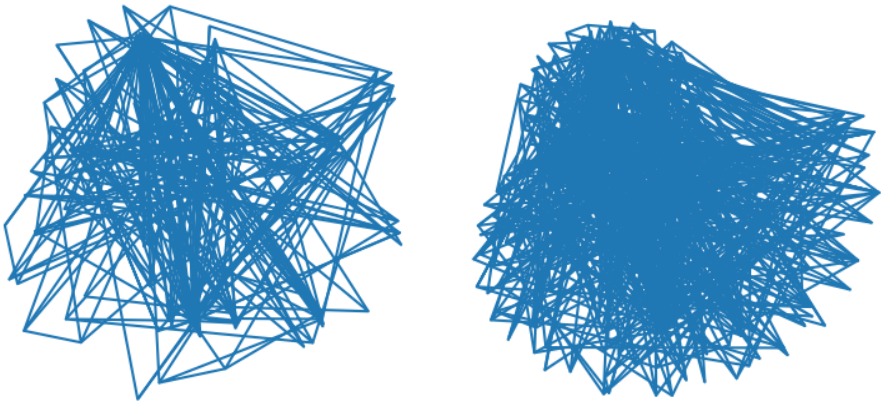
## The order in control points and evaluation points



(a) Original control polygon of duck model

(b) Evaluation polygon generated by original control points

Figure 2.6: Control polygon and evaluation polygon of original control points



(a) Shuffled control polygon of duck model

(b) Evaluation polygon generated by shuffled control points

Figure 2.7: Control polygon and evaluation polygon generated by shuffled control points

As described previously, control points play a vital role in defining the shape of a NURBS surface. Variation in position or order of the control points should directly change the topology of its corresponding control polygon. According to Equation 2.1, control points $P_{i,j}$ is involved with basis functions $N_{i,p}(u)$ and $N_{j,q}(v)$, so they only influence evaluation points within the parametric domain range of $(u_i, u_{i+1}]$ and $(v_i, v_{i+1}]$. Therefore, if the order in the control point set changes, related evaluation points on the surface should be changed by their new corresponding control points accordingly. This relationship can be visualized using the example duck model. Figure 2.6 shows a control polygon generated from original control points of the duck model, along with the parametrically distributed evaluation points. However, as seen in Figure 2.7, the shuffled control polygon is still ordered, but not like the original one, which leads to an unstructured evaluation point cloud.

Therefore, by examining the order in control points on both control polygon and corresponding parametric sampled points, it becomes evident the ordering of control points is critical in NURBS surface reconstruction. This is particularly crucial when performing unsupervised learning methods, since under unsupervised settings, it is unlikely to have any knowledge regarding the true order in evaluation points which are controlled by the control polygon. We will elaborate on this in the following chapters.

### Special cases

From the above introduction, NURBS is short for non-uniform rational basis spline. Non-uniform refers knot vector is not uniform. Rational indicates the weight of each control point is not the same. If knot vectors are uniform, then the NURBS surface is a uniform rational B-spline surface(URBS). If the weights are all one, which means all control points have the same influence on the surface, then the NURBS surface is a non-rational B-spline surface (NUBS). In common, we usually consider uniform B-spline surface (UBS) as our reconstruction target, from which we only have to train the control points.

### Control points and weights format: ctrlpts, weights and cptw

In the previous subsection, we have introduced the concept of control points and weights. In this subsection, we will introduce the format of control points and weights. Files in ctrlpts format store control points as follows:

**Listing 2.4: ctrlpts file format**

```
...
1.92379 0.0465361 -0.535702
1.90747 0.121179 -0.535702
1.57358 0.236742 -0.535702
...
```

Each row in Listing 2.4 represents a control point $(x, y, z)$.

Files in weights format store weights as follows:

**Listing 2.5: weights file format**

```
...
1,1,1,1,
0.5,1,1,1,
1,1,0.5,1,
1,1,1,1,
...
```

Each element separated by a comma in Listing 2.5 represents a weight vector and each weight corresponds to the control point in the same order. While files in cptw format store control points and weights as follows:

**Listing 2.6: cptw file format**

```
    1.0,0.0,0.0,1.0;0.7071,0.7071,0.0,0.7071;0.0,1.0,0.0,1.0
    1.0,0.0,1.0,1.0;0.7071,0.7071,0.7071,0.7071;0.0,1.0,1.0,1.0
    % Long content that extends beyond the available space
```

Listing 2.6 shows how each weight is attached to its matching control point.

## 2.6 NURBS-Python(geomdl)

**Listing 2.7: Code snippet for duck model**

```
import os
from geomdl import NURBS
from geomdl import multi
from geomdl import exchange
from geomdl import utilities
from geomdl import compatibility
from geomdl.visualization import VisMPL
import numpy as np
from scipy.optimize import minimize
import torch

def read_weights(filename, sep=","):
  try:
    with open(filename, "r") as fp:
      content = fp.read()
      content_arr = [float(w) for w in (''.join(content.split())).split(sep)]
      return content_arr
  except IOError as e:
    print("An error occurred: {}".format(e.args[-1]))
    raise e

# Fix file path
os.chdir(os.path.dirname(os.path.realpath(__file__)))

# duck1.nurbs
# Process control points and weights
d2_ctrlpts = exchange.import_txt("duck1.ctrlpts", separator=" ")
d1_weights = read_weights("duck1.weights")
d1_ctrlptsw = compatibility.combine_ctrlpts_weights(d2_ctrlpts, d1_weights)

# Create a NURBS surface
duck1 = NURBS.Surface()
duck1.name = "body"
duck1.order_u = 4
duck1.order_v = 4
duck1.ctrlpts_size_u = 14
duck1.ctrlpts_size_v = 13
duck1.ctrlptsw = d1_ctrlptsw
duck1.knotvector_u = [-1.5708, -1.5708, -1.5708, -1.5708, -1.0472, -0.523599, 0,
    0.523599, 0.808217,
            1.04015, 1.0472, 1.24824, 1.29714, 1.46148, 1.5708, 1.5708, 1.5708,
                1.5708]
duck1.knotvector_v = [-3.14159, -3.14159, -3.14159, -3.14159, -2.61799, -2.0944,
    -1.0472, -0.523599,
            6.66134e-016, 0.523599, 1.0472, 2.0944, 2.61799, 3.14159, 3.14159,
                3.14159, 3.14159]

duck1.evaluate()
# Visualization configuration
duck1.vis = VisMPL.VisSurface(ctrlpts=True, legend=False)

# Render the ducky
duck1.render(cpcolor='lightgreen', evalcolor='violet', filename="duck1.png", plot=
    False)
```

NURBS-Python[7] is a pure Python, object-oriented B-spline and NURBS library for Python. The library, inspired by the paper [BK19], provides extensive methods for B-splines, NURBS curves, and NURBS surfaces. The library can work along with other libraries such as NumPy and SciPy [VGO+20]. Moreover, it also provides visualization functionality via Matplotlib, VTK [SML06] and other libraries. The code, modified from the official geomdl github repository, is shown in Listing 2.7. In Listing 2.7, the type of NURBS is declared as a NURBS surface. In the following, the parameters of a NURBS surface can be set through utility functions or file upload. Next, the surface is computed and sampled points are evaluated given these parameters. Finally, the surface is plotted using built-in visualization tools.

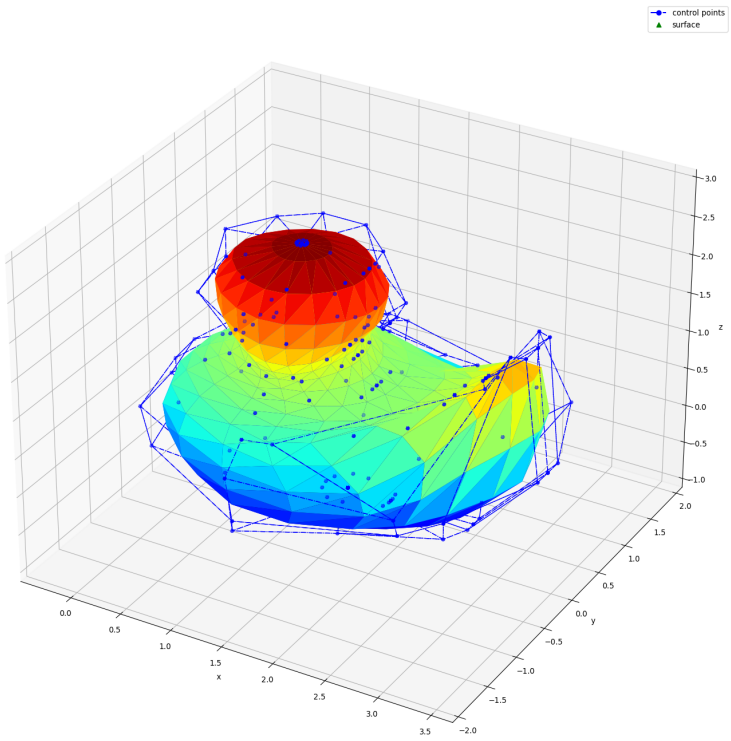



Figure 2.8: Duck NURBS generated by geomdl

As shown in Figure 2.8, blue points represent control points, which form a control polygon covering the colorful duck mesh. With geomdl library, we can generate a NURBS surface with provided parameters and it is extremely useful for our evaluation process.

## 2.7 SubDivision surface

A subdivision surface[8] [Ma05] (SubD surface) utilizes a specified algorithm to generate a curved surface which is represented by multiple coarser polygon meshes. The underlying surface can be decided by the coarse mesh. These outer meshes are usually called control cages, which is similar to control polygons which are generated by control points of NURBS surfaces. However, Catmull-Clark algorithm [CC78] is always applied to generate outer meshes, which runs iteratively to subdivide each polygon face into smaller faces to approximate the underlying surface.

[7] https://github.com/orbingol/NURBS-Python
[8] https://en.wikipedia.org/wiki/Subdivision_surface

## 2.8 Point cloud comparison

### L2 norm

L2 loss aims to minimize the mean square error(MSE[9]) between two point clouds. It is computed by the following equation:

$$\mathcal{L}_{L2}(X,Y) = \frac{1}{N}\sum_{i=1}^{N} \|x_i - y_i\|_2^2 \tag{2.6}$$

where $X$ and $Y$ are two point clouds with the same size. L2 requires two point clouds with the same size. Shuffling the order of points in either point cloud will change the result of L2 loss since one-to-one correspondence relationship is disordered.

### Chamfer

Chamfer distance[10] is a metric to measure the similarity between two point clouds. It calculates the distance between each point either in point cloud $X$ or $Y$ to the nearest point in the other point cloud, which is computed by the following equation:

$$\mathcal{L}_{Chamfer}(X,Y) = \sum_{x\in X}\min_{y\in Y}\|x-y\|_2^2 + \sum_{y\in Y}\min_{x\in X}\|x-y\|_2^2 \tag{2.7}$$

where $X$ and $Y$ are two point clouds. The Chamfer does not require two points clouds with the same size. Shuffling the order of points in either point cloud will not change the result of Chamfer distance.

### Hausdorff

Hausdorff distance[11] is another metric to measure the similarity between two point clouds. However, it will find the maximum distance between any point either in point cloud $X$ or $Y$ to the nearest point in the other point cloud. It is computed by the following equation:

$$\mathcal{L}_{Hausdorff}(X,Y) = \max(\max_{x\in X}\min_{y\in Y}\|x-y\|_2, \max_{y\in Y}\min_{x\in X}\|x-y\|_2) \tag{2.8}$$

where $X$ and $Y$ are two point clouds. Like Chamfer distance, the Hausdorff distance allows for the comparison of point clouds without requiring them to be of the same size, while shuffling the order of points in either point cloud will not change the result of Hausdorff distance.

### Laplacian

Laplacian matching Loss, adopted in ParSeNet [SLK+20], is a specific loss function for B-splines to enable the smoothness of the surface, which aims to minimize the difference between Laplacian operators applied on two point clouds. It is computed by the following equation:

$$\mathcal{L}_{Laplacian}(X,Y) = \frac{1}{N}\sum_{i=1}^{N} \|L(x_i) - L(y_i)\|_2^2 \tag{2.9}$$

where $X$ and $Y$ are two point clouds with the same size. $L$ is the Laplacian operator.

[9] https://en.wikipedia.org/wiki/Mean_squared_error
[10] https://pdal.io/en/latest/apps/chamfer.html
[11] https://en.wikipedia.org/wiki/Hausdorff_distance

## 2.9 Related softwares

### MeshLab

MeshLab [CCC+08] is an open source software for 3D mesh processing. It can be used to display and edit mesh data and point cloud. For instance, it can be used to remove noise from point clouds, or to sample point clouds from objects. In this thesis, we will use MeshLab to retrieve sampled point clouds from dedicated objects using Poisson disk sampling and evaluate the alignment of point clouds.

### Rhinoceros 3D

Rhinoceros[12] (abbreviated Rhino) is a commercial 3D computer graphics and computer-aided design application software. Rhino has good support for NURBS surfaces, subdivision surfaces as well as many free-form surfaces. Since we focus on exploring the magic of NURBS surfaces, we use Rhino 7(latest version) to create some NURBS surfaces to study their properties. With these NURBS surfaces, we can extract their parameters from Rhino 7 by using some scripts and commands.

[12]https://en.wikipedia.org/wiki/Rhinoceros_3D

# 3 Related Work

In the field of 3D reconstruction, there are multiple ways to represent the shape of a 3D object. The most common way is to use polygon mesh representation, which uses a composition of vertices, edges, and faces for defining the shape of a polyhedral[1] object. There are numerous papers focusing on reconstructing objects using mesh representation, like the poco model proposed by Boulch and Marlet [BM22], which uses a modified implicit neural network to retrieve surfaces from point clouds.

However, this approach also reveals some drawbacks. The main issue is the training process is typically time-consuming, while the trained model is not generalized enough. On one hand, it usually takes a long time to train a model for polygon mesh reconstruction since the input point is very large if we need to obtain high resolution meshes. Besides, we usually need to train a pre-trained model with tons of data to fit our needs, which is also time-consuming. On the other hand, we may prepare several models for different kinds of meshes. Like the poco model mentioned above, it can only process watertight meshes, while we require alternative models to process non-watertight meshes. However, NURBS surfaces can handle these issues due to their natural properties. With NURBS surface reconstruction method, we do not need several models since NURBS surfaces can handle data from watertight and non-watertight meshes. Moreover, NURBS surface reconstruction is much faster because NURBS mainly focuses on parameters rather than a large point cloud. Therefore, NURBS surfaces, which utilize B-splines to mathematically model surfaces, provide another way for surface reconstruction from parameters. NURBS surfaces are able to restructure any shape, provided that the parameters are known.

Sharma et al. propose a trainable deep network called ParSeNet. ParSeNet takes a 3D point cloud as the input by segmenting it into multiple patches as B-spline surfaces and basic primitives. Once the initial step is finished, the network is able to output surfaces with high fidelity via SplineNet module and primitive fitting. SplineNet is another neural network aiming to perform B-spline surface fitting under a supervised learning method. Besides, for open and closed splines, two pre-trained models should be prepared beforehand respectively. The corresponding codebase[2] provides extensive utility functions.

Prasad et al. propose a method to reconstruct NURBS surfaces from point clouds by surface parameterization [DBS+22] to avoid the limitation illustrated above. In this paper, they implement the NURBS-Diff module, which is to train a NURBS-based point cloud to fit the ground truth one. The module mitigates the issue mentioned above by working without any pre-training and enhancing generalization.

Our study is based on the work by Prasad et al.. With the help of some key utility functions implemented in ParSeNet, we are able to explore the potential of the NURBS-Diff module.

Since the papers mentioned above mainly focus on supervised learning methods for parametric surface reconstruction from structured point clouds, which indicates the methods require the ground truth parameters. For instance, ParSeNet needs to segment input point cloud before surface fitting. Moreover, the surfaces to be fitted are also some simple shapes such as cones or cylinders. While the NURBS-Diff module is able to handle more complex models like a duck, it still needs to know the parameters beforehand. However, in practice, it is not always possible to obtain these parameters. The authors who propose the NURBS-Diff module also discuss a method for point cloud reconstruction but have not shared their code for this approach. Our work, instead, building on top of their supervised pipeline, is capable of reconstructing complex NURBS-based point clouds from unstructured point clouds without point cloud segmentation. Our proposed method aims to deal with complex models in the real world rather than simple objects for experiment.

[1] https://en.wikipedia.org/wiki/Polyhedron

[2] https://github.com/Hippogriff/parsenet-codebase

## 3.1 Related papers

### 3.1.1 ParSeNet

As mentioned at the beginning of this chapter, ParSeNet is able to restructure B-spline surfaces and primitive surfaces. Reconstruction of primitive surfaces is omitted in our study since it is not our focus. SplineNet, which is used to process B-spline surface fitting, is the core component of ParSeNet. SplineNet inputs segmented point cloud and its corresponding control points which are extracted from SplineDataset [SLK$^+$20], following that, it generates a control polygon with fixed size as the training control points. Since the reconstruction loss is related to flips or swaps of the control polygon, the network adopts Chamfer loss and regression loss for permutations of control points. Besides, Laplacian loss is leveraged to compute the difference between sampled points and predicted point cloud.

Nonetheless, the network still exists some drawbacks. The main shortcoming is the network adopts a supervised learning method since it requires a ground truth control points grid to evaluate regression loss and Laplacian loss. But in most cases, it is impossible for us to obtain the underlying parameters of a mesh. Typically, we can only get the mesh or point cloud instead. Besides, ParSeNet requires segmenting input point cloud in the initial step and B-splines as well as primitives are not complex. Anyway, it offers a feasible approach for reconstructing B-spline surfaces and provides methods that can be used during the training process.

### 3.1.2 NURBS-Diff

NURBS surfaces are defined by control points, weights, knot vectors, and degrees. Knot vectors and degrees calculate the basis functions according to Equation 2.3. With the knowledge of Equation 2.1, it is possible to create a neural network on top of it. If the surface parameters can be retrieved, the surface can be immediately reconstructed. However, applying any deep learning method to NURBS is quite challenging, as the surfaces are always not differentiable. According to Equation 2.3, the gradients of knot vectors cannot always be computed. Specifically, since basis functions are defined recursively, therefore, in order to compute the gradient of knot vectors, it is inevitable to compute the derivative of 0-degree basis functions $N_{i,0}(u, v)$ and $N_{j,0}(u, v)$. However, these 0-degree basis functions are non-differentiable at the endpoints, which brings out the idea of the NURBS-Diff module, a module for NURBS deep learning [DBS$^+$22].

In the paper written by Prasad et al., they introduce a method to compute the gradient of NURBS surfaces. The key point is to replace the original 0-degree basis functions with the subtraction of two variant delta functions. However, the gradient of the NURBS surface can be computed as 0 everywhere and it is still not differentiable at endpoints. To solve this issue, they approximate the derivative using Gaussian smoothing to replace delta functions with Gaussian functions. In this way, the gradient of NURBS surfaces can be computed everywhere. In addition, they implement surface point evaluation from forward propagation based on 2.1, which allows for a parametric sampling from the surface. The generated point cloud is also called evaluation points. The order in the point cloud corresponds to the order in 2D parametric space.

The paper aims to perform NURBS surface fitting. Given dedicated resolution, which is the number of points sampled on $u$ and $v$, evaluation points are sampled from 2D parametric space uniformly. By minimizing the loss between training point cloud and ground truth evaluation points, the trained point cloud should align with ground truth one finally.

Nevertheless, the paper does have some limitations. On one hand, the work is limited by its supervised method. Evaluation points generated via forward propagation build on top of the known parameters. On the other hand, the training loss for point cloud reconstruction proposed in this paper is a combination of Chamfer distance, Hausdorff distance, and Laplacian loss. However, the related implementation is not provided in their codebase[3], thus it is impossible for us to reproduce the results. To address these issues, we tweak the original supervised pipeline using the NURBS-Diff module to further implement unsupervised pipelines.

[3] https://github.com/idealab-isu/NURBSDiff/tree/master

## 3.2 Related pipelines

### Surface fitting using NURBS-Diff

The pipeline of surface fitting using the NURBS-Diff module proposed in chapter 3.1.2 from the paper [DBS+22] includes several steps. First, there is the loading of surface parameters. The surface parameters should be the ground truth, with which it is capable to generate a ground truth evaluation point cloud. Following that, similarly, with initialized parameters rather the ground truth, the training point cloud is generated. Afterwards, the NURBS-Diff module is leveraged to compute the gradient of the NURBS surface. The loss is computed by comparing trained and ground truth point clouds. Finally, the loss is backpropagated to update the parameters which need to be trained. The pipeline is shown in Figure 3.1.

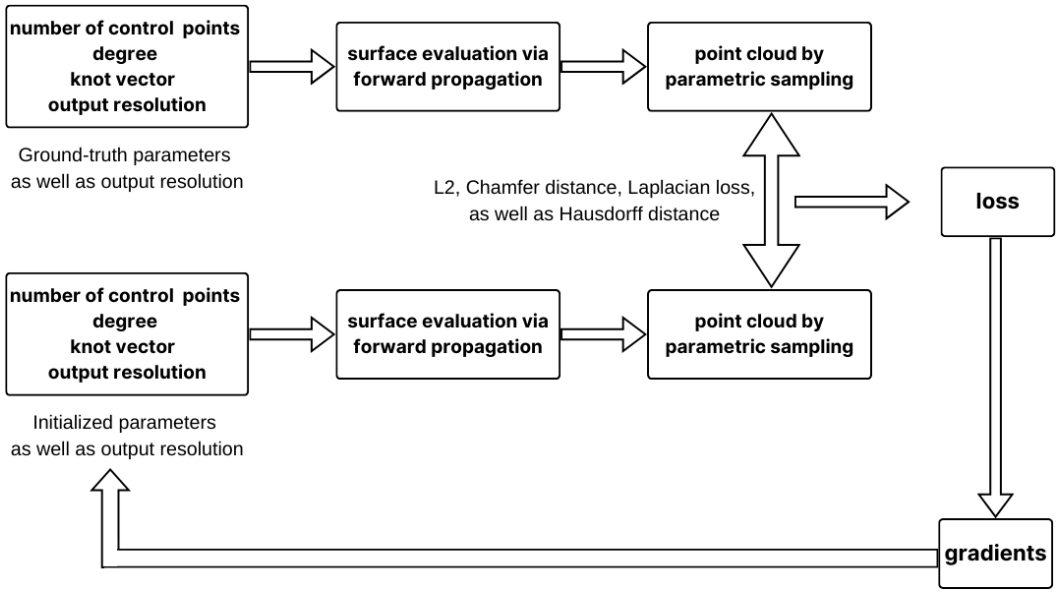


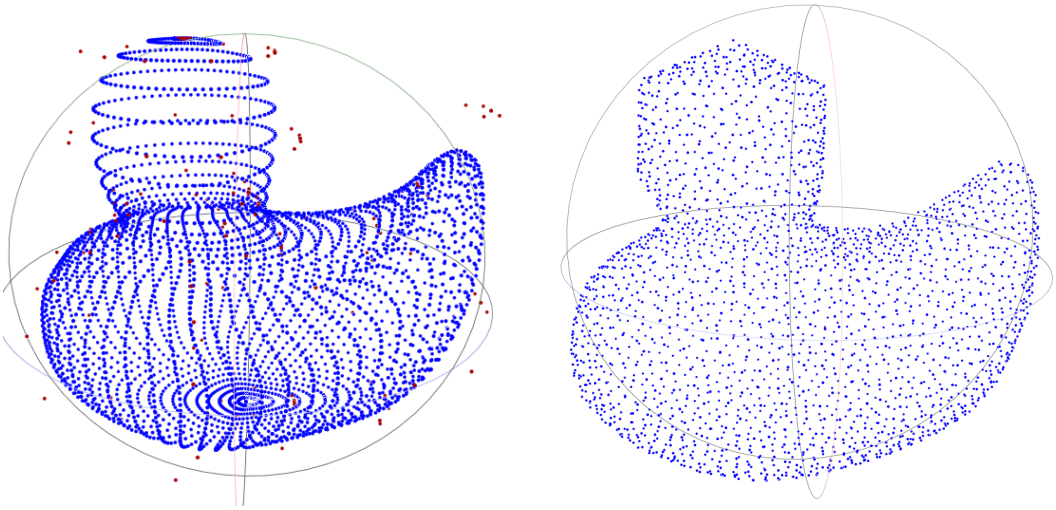

Figure 3.1: Pipeline of surface fitting using NURBS-Diff

### Loading parameters of NURBS surface and generating ground truth point cloud

As mentioned above, the first step is to load the parameters of the target NURBS surface. Using these known parameters, a ground truth point cloud can be sampled from the surface via forward evaluation. As shown in Figure 3.2(a), the duck point cloud is generated parametrically from the target NURBS surface. It is clear this point cloud has a beautiful spiral pattern, which is quite different from another duck point cloud sampled by Poisson disk method shown in Figure 3.2(b). This is because points in the former are extracted through the uniformly sampled 2D parametric space. Given Figure 3.3, sampling the parametric domains $u$ and $v$ at discrete intervals uniformly is analogous to uniform sampling from a 2D texture which is mapped to a 3D object.

Table 3.1: Illustration of uniformly sampled $u$ and $v$

| $u$ \ $v$ | 0 | 0.2 | 0.4 | 0.6 | 0.8 | 1 |
|---|---|---|---|---|---|---|
| 0 | (0,0) | (0, 0.2) | (0, 0.4) | (0, 0.6) | (0, 0.8) | (0, 1.0) |
| 0.2 | (0.2,0) | (0.2, 0.2) | (0.2, 0.4) | (0.2, 0.6) | (0.2, 0.8) | (0.2, 1.0) |
| 0.4 | (0.4,0) | (0.4, 0.2) | (0.4, 0.4) | (0.4, 0.6) | (0.4, 0.8) | (0.4, 1.0) |
| 0.6 | (0.6,0) | (0.6, 0.2) | (0.6, 0.4) | (0.6, 0.6) | (0.6, 0.8) | (0.6, 1.0) |
| 0.8 | (0.8,0) | (0.8, 0.2) | (0.8, 0.4) | (0.8, 0.6) | (0.8, 0.8) | (0.8, 1.0) |
| 1.0 | (1.0,0) | (1.0, 0.2) | (1.0, 0.4) | (1.0, 0.6) | (1.0, 0.8) | (1.0, 1.0) |



(a) The point cloud sampled from duck model by parametric sampling

(b) The point cloud sampled from duck model by Poisson disk sampling

Figure 3.2: Point clouds sampled from duck model by different means

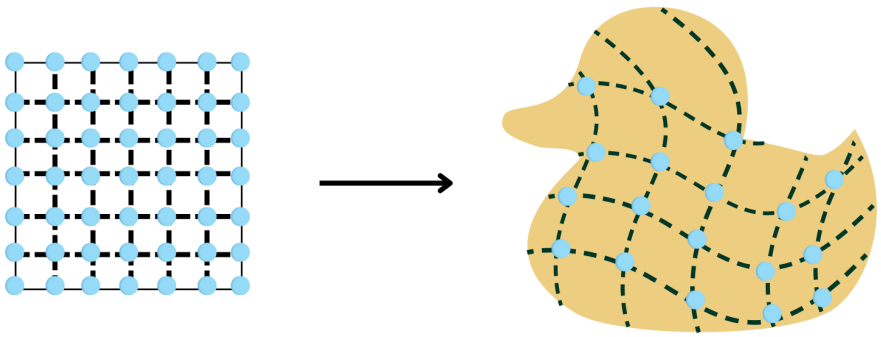


Figure 3.3: Mapping from a subset of $R^2$ to $R^3$. The neighbor points in the domain of $R^2$ should be still adjacent in $R^3$ to make surface continuous.

Furthermore, if we aim to sample 36 points uniformly from a given NURBS surface with known parameters, $u$ and $v$ in 2D parametric space should be divided into 5 equal slices. The $(u, v)$ pairs, which are listed as follows in Table 3.1, form a $6 \times 6$ grid. The points are ordered in this grid, while they provide the parametric locations $(u, v)$ via the forward propagation. According to Equation 2.1, 36 ordered points are generated in a parametric sampling way from the underlying surface.

This process of generating sampled point clouds on parametric domains is referred to as parametric sampling in this thesis. Parametric sampling is the key component of the NURBS-Diff module, which generates point clouds with a specific NURBS pattern. We will reuse the concept therein when generating such point clouds in the following sections and chapters.

### Generating point cloud from initialized (random) parameters

The initialization of the parameters for the NURBS surface is an important step as well. In their implementation, weights are initialized to be uniform, and control points are set to be random. As introduced in section 2.5, NURBS knot vectors are non-decreasing real numbers where only interior knots can change freely. Considering the restriction of knot vectors, the initial knot vectors along $u$ and $v$ direction are set to a vector of ones, while the boundary takes repeated zero and one. With these initial parameters, it is capable of sampling a point cloud by performing forward propagation of the NURBS-Diff module. As known from the previous subsection, the point cloud also adheres to parametric sampling.

### Training loop with NURBS-Diff module

With ground truth point cloud and initialized point cloud, they train the parameters with the module NURBS-Diff. The module is designed to compute the gradients of NURBS surfaces with regard to its parameters. The module is implemented in Pytorch for deep learning and C++ for acceleration. The algorithm and methods used for generating NURBS surfaces follows the implementation of geomdl, though geomdl only works with pure Python. Instead, the NURBS-Diff module takes advantage of Pytorch and leverages C++, which performs forward and backward propagation efficiently.

During backward propagation, the ground truth point cloud is compared to the generated point cloud through L2 loss. The L2 loss is selected for training the supervised pipeline since it requires two point clouds of equal size, which would satisfy the need to compare corresponding pairs in point clouds. Consequently, the parameters will be updated once the gradients are computed based on L2 loss.

### Displaying point clouds

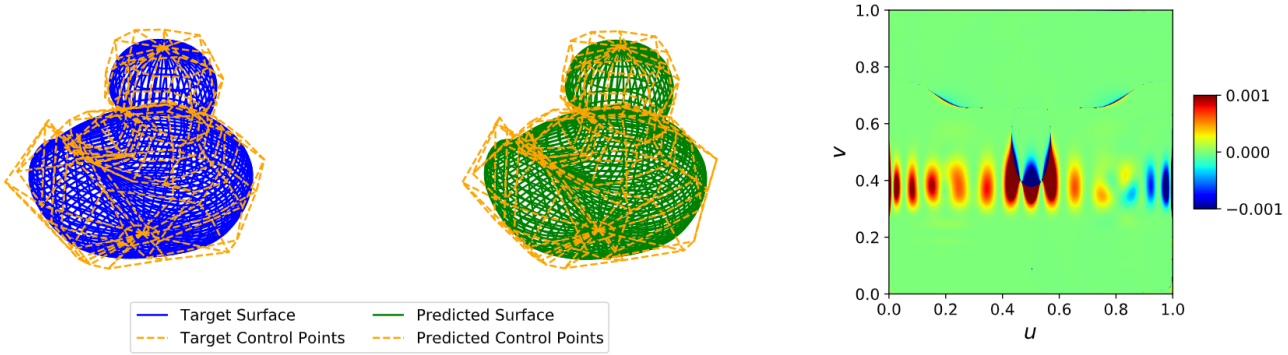



Figure 3.4: Supervised learning result of duck model

Finally, the authors display input and output point clouds as well as corresponding control polygons using Matplotlib and save them to a PDF file for later use.

Take the duck model as an example, they load all parameters of the model and compute the target point cloud by parametric sampling with regard to the resolution. In the next step, they generate a point cloud with initial parameters by parametric sampling as well. This allows the same parametric order of $(u, v)$ pairs in two point clouds. After the training loop, the parameters and output point cloud are also stored. An error map is displayed in Figure 3.4. As it can be observed from the error map, the generated wireframe with color green is aligned with the ground truth one with color blue. Also, the control polygons are very similar with acceptable error. The error map on the right shows the error in each $(u, v)$ pair. The bulk part is rendered with color green, therefore they draw the conclusion with the trained parameters, the generated surface performs well with the alignment of the ground truth surface.

Despite this, they do not use geomdl to validate the result. Instead, we will leverage the package to validate the result in the following chapters.

# 4 Problem statement

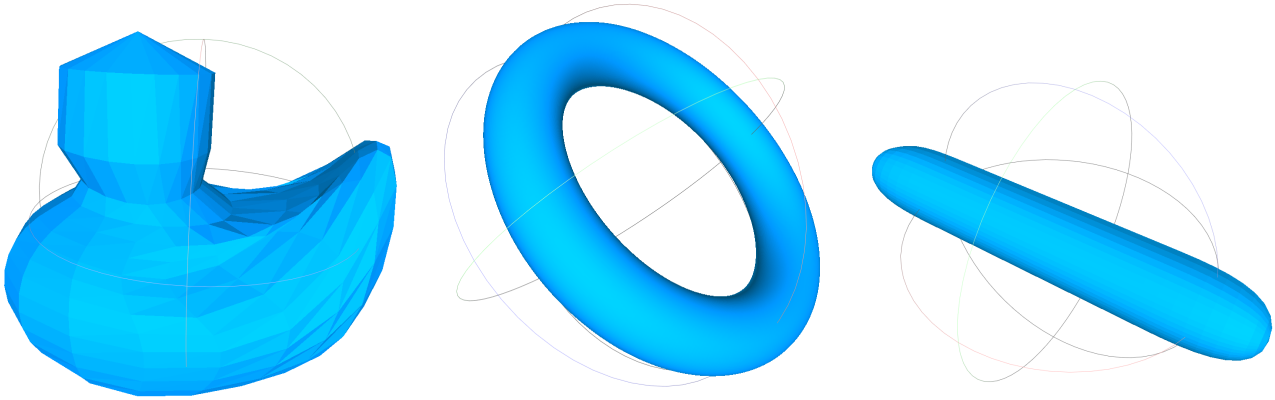
As discussed in the previous chapter, the paper [DBS+22] proposes a supervised pipeline which utilizes NURBS-Diff module to train the parameters of NURBS surfaces. However, the pipeline in the paper is limited to several aspects. First, it only works within the scope of NURBS surfaces whose parameters are already known. second, it is incapable of dealing with complex shapes, like organic free-from shapes.

Therefore, the thesis aims to devise an unsupervised method to perform the task of 3D object reconstruction directly from point clouds. Specifically, the objectives are:

- Verify the supervised pipeline proposed in the paper [DBS+22], analyze under what conditions it is effective.
- Implement unsupervised methods for NURBS objects(or non-NURBS objects) using the NURBS-Diff module to retrieve parameters without any supervision. The performance will also be validated by geomdl library.
- Extend the unsupervised approach to non-NURBS objects by applying some alternative algorithms and validating reconstructed parameters with geomdl library.

The pipelines we implemented are illustrated extensively in Chapter 5. Prior to the introduction to our implementation, there are some concepts and algorithms to clarify.

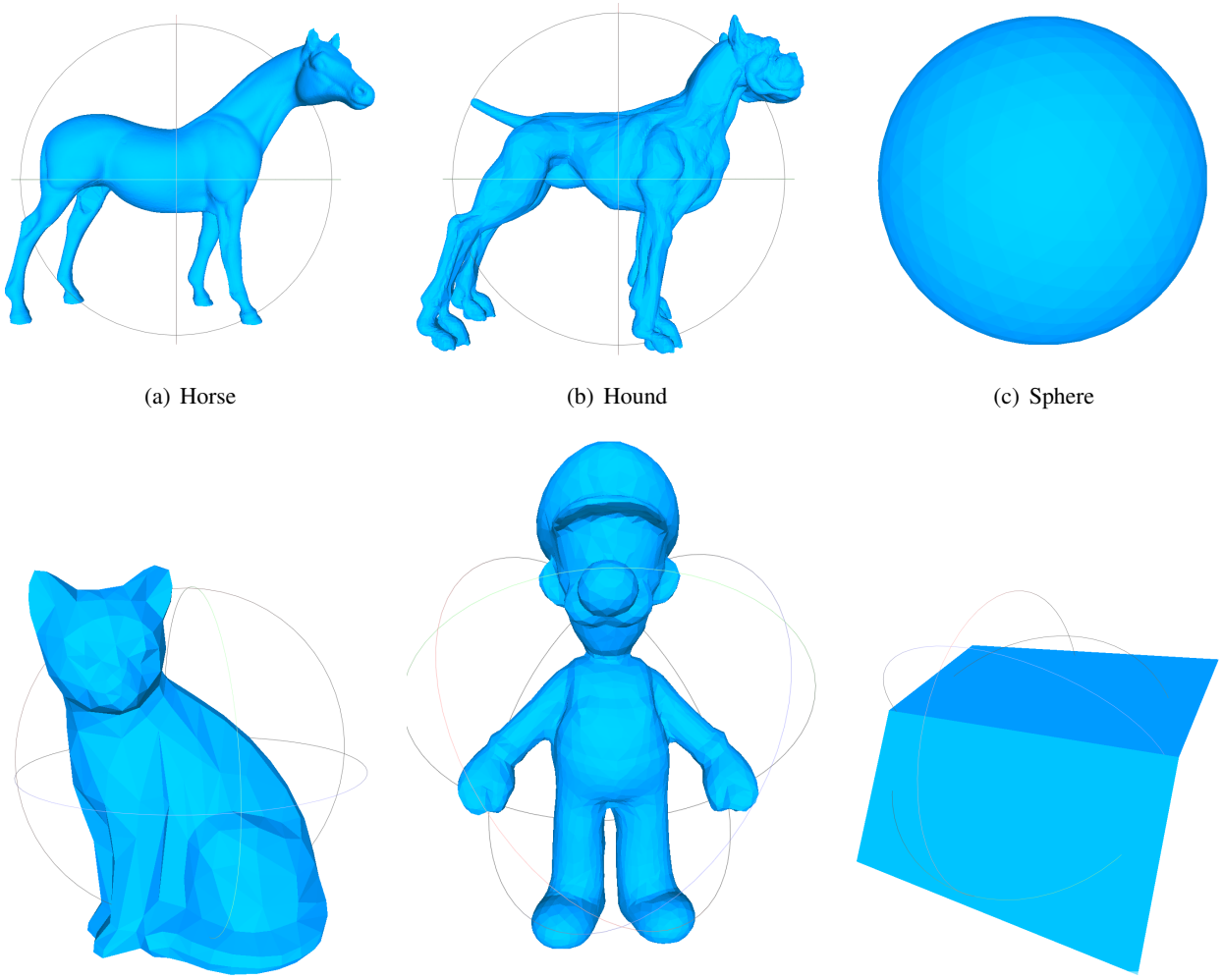
## 4.1 Data



(a) Duck (b) Torus (c) Cylinder

Figure 4.1: NURBS-based meshes

The NURBS objects we used are very simple. The duck model and some generated NURBS surfaces by Rhino, shown in Figure 4.1, are used mainly for supervised learning. Although we do indeed obtain the parameters of these objects, we can still input vertices of these objects into unsupervised pipelines.



(a) Horse (b) Hound (c) Sphere

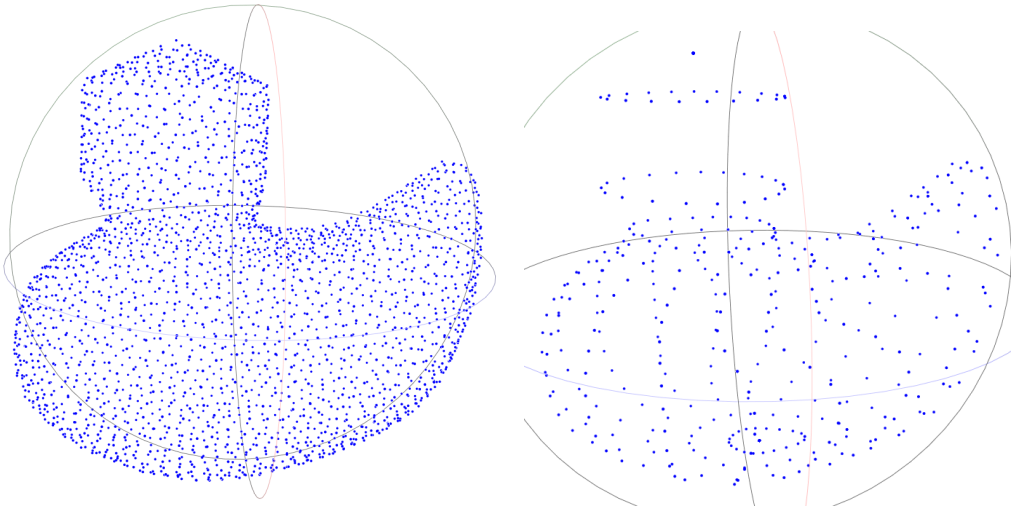
(d) Cat (e) Luigi (f) Plane

Figure 4.2: Normal meshes

Besides, some mesh-based objects, as displayed in Figure 4.2, are also used for unsupervised learning. All the files are in OFF format, so they can be pre-processed in the same manner.

Point clouds can be easily sampled from these meshes either via scripts or MeshLab. The sampled point clouds, some of which are presented in Figures 4.3 and 4.4, are saved in OFF format. The size (or resolution) of point clouds can be adjusted. Note Subfigure 4.3(b) shows a point cloud extracted from the vertices of the duck model, which keeps the NURBS pattern. Additionally, These vertices are evaluation points of its underlying NURBS surface, which can be considered as the point cloud generated from forward propagation of the NURBS-Diff module.

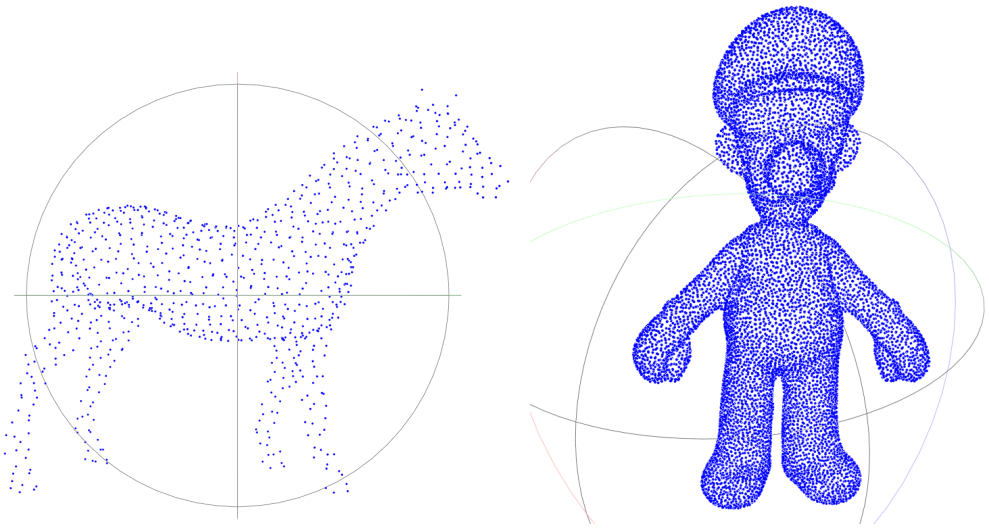
These point clouds are used as the input for our different pipelines.



(a) The point cloud from duck model by Poisson disk sampling

(b) The vertices extracted from duck mesh

Figure 4.3: Sampled point clouds I



(a) The point cloud sampled from horse mesh by Poisson disk sampling

(b) The point cloud sampled from luigi mesh by Poisson disk sampling

Figure 4.4: Sampled point clouds II

## 4.2 NURBS-Diff module

This section explains the mechanism behind the NURBS-Diff module with a detailed illustration. This is the key module of our unsupervised learning pipelines. We start with forward propagation by explaining how it is capable of generating a NURBS-based point cloud via parametric sampling. Subsequently, we dive into the backward propagation by discussing its underlying algorithm. Finally, we introduce the files in the module and analyze their functionalities.

### 4.2.1 Forward propagation

To do the forward propagation with known parameters, first, the resolution of the input point cloud is defined to be $n_{grid} \times m_{grid}$. Accordingly, a mesh grid of $u_{grid} \times v_{grid}$ is uniformly sampled in both directions into $n_{grid} - 1$ and $m_{grid} - 1$ slices. The forward algorithm is illustrated as follows.

**Algorithm 1** Forward algorithm: surface evaluation algorithm

**Input:** $U, V, P, W, n_{\text{grid}}, m_{\text{grid}}$
**Output:** $S$
Initialize $u_{\text{grid}} \times v_{\text{grid}}$ uniformly sample from [0, 1]
Initialize: $S \to 0$
**for** $j = 1$ to $m_{\text{grid}}$ points in parallel **do**
  **for** $i = 1$ to $n_{\text{grid}}$ points in parallel **do**
    Compute $u_{\text{span}}$ and $v_{\text{span}}$ for the corresponding $u_i$ and $v_j$ using knot vectors $U$ and $V$
    Compute basis functions $N_i$ and $N_j$ basis functions using $u_{\text{span}}$ and $v_{\text{span}}$ and knot vectors $U$ and $V$
    Compute sampled point $S(u_i, v_j)$
    Store $u_{\text{span}}$, $v_{\text{span}}$, $N_i^p$, $N_j^q$
  **end for**
**end for**

The algorithm, which follows the construction of NURBS surfaces, is not difficult to understand. The surface evaluation algorithm establishes the one-to-one mapping relationship between parametric coordinate points and their 3D locations on the NURBS surface. For any point $(n_i, m_j)$, there is a definite matching point $(u_i, v_j)$. $u_{span}$ and $v_{span}$ in the algorithm are the variation of $u_{grid}$ and $v_{grid}$. They have the same size, however, in consistence with the definition of NURBS basis functions, $u_{span}$ and $v_{span}$ range in value from the degree $d + 1$ to $d + len(U)$ along the $u$ direction and analogously along $v$. This is originally implemented in *evaluators.py*[1] from geomdl github repository. With stored knot vectors and basis functions, evaluation points can be sampled using Equation 2.1. It is also important to note a point on the surface does not have any relationship with other points according to Equation 2.1. The position of a point only depends on its located span on knots as well as its corresponding control point. This property enables the parallel computation among points, which significantly accelerates when implemented in C++. Finally, $u_{span}$, $v_{span}$ and basis functions are stored for backward propagation.

### 4.2.2 Backward propagation

With the replacement of sign functions with Gaussian functions, the derivative of parameters can be computed as the following backward algorithm illustrates.

In backward propagation, although the point cloud is discrete, however, by leveraging the differentiable NURBS surface formula, gradients can be computed with regard to each sampled point $(u_i, v_j)$, therefore, it is capable of updating the parameters to perform another iteration.

[1] https://github.com/orbingol/NURBS-Python/blob/master/geomdl/evaluators.py

**Algorithm 2** Backward algorithm: gradient computation

**Input:** $S'$, $u_{span}$, $v_{span}$
**Output:** $P'$, $W'$
Initialize: $P' \rightarrow 0$
Initialize: $W' \rightarrow 0$
**for** $j = 1$ to $m_{grid}$ **do**
    **for** $i = 1$ to $n_{grid}$ **do**
        Retrieve $u_{span}$, $v_{span}$, $N_i^p$, $N_j^q$, $S(u, v)$
        **for** $r = 0$ to $p + 1$ **do**
            **for** $h = 0$ to $q + 1$ **do**
                $P'[u_{span} + r, v_{span} + h] \leftarrow S_{p_{i,j}}(u_i, v_j)$
                $W'[u_{span} + r, v_{span} + h] \leftarrow S_{w_{i,j}}(u_i, v_j)$
                $U'[u_{span} + r] \leftarrow S_{u_{span}+r}(u_i, v_j)$
                $V'[v_{span} + h] \leftarrow S_{v_{span}+h}(u_i, v_j)$
            **end for**
        **end for**
    **end for**
**end for**

### 4.2.3 An overview of NURBS-Diff module

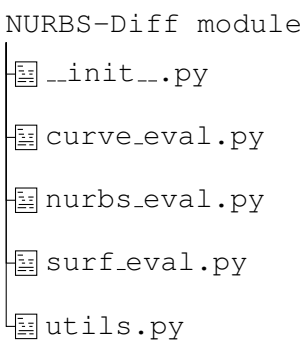



Figure 4.5: The structure of NURBS-Diff module

The structure of the NURBS-Diff module is illustrated in Figure 4.5.

*__init__.py* is to declare this package as a module.

*curve_eval.py* is the file to compute evaluation points on NURBS curves. Besides, it can be used for backward propagation by autograd functions.

*surf_eval.py* is the file to compute evaluation points on target NURBS surfaces. Initial parameters are provided to compute $u_{span}$, $v_{span}$, and basis functions $Nu\_uv$ and $Nv\_uv$. Upon inputting initialized control points, the forward evaluation is performed by Equation 2.1 with dedicated output resolution.

*nurbs_eval.py* is the file to perform forward and backward on training NURBS surfaces. As stated before, this file uses a similar approach for forward evaluation. In backward propagation, with stored knot vectors and basis functions, it is capable of computing gradient with regard to parameters by means of Gaussian smoothing.

*utils.py* is the utility functions migrated from geomdl.

In this thesis, we mainly use *surf_eval.py* for target NURBS surface evaluation and *nurbs_eval.py* for the whole training process.

## 4.3 Point to surface distance

In paper [PGK02], Pauly et al. verify the point-to-surface distance can be approximated by the distance between the point and the closest point on the surface if there are enough sampled points from the surface. This is significant because it is always harsh to compute the distance between a point and a surface. However, computing the distance between points is much easier. We also verify this conclusion using the following code with the help of geomdl and SciPy.

**Listing 4.1: Testing the approximation algorithm**

```
...
from scipy.optimize import minimize

duck1.sample_size = 100
duck1.evaluate()
# This function calculates the distance between a given point in 3D space and the
# closest point on a given surface in 3D space using scipy.optimize.minimize.
def point_surface_distance(point, surface):
    def objective(uv):
        pt_on_surf = surface.evaluate_single(uv)
        return np.linalg.norm(np.array(point) - np.array(pt_on_surf))
    bounds = [(0, 1), (0, 1)]
    result = minimize(objective, x0=[0.5, 0.5], bounds=bounds)
    return result.fun, result.x

def point_point_distance(point, point_set):
    dists = np.linalg.norm(point_set - point, axis=1)
    return np.min(dists)

err = 0
for i in range(1000):
    point = np.random.rand(3)
    dist, uv = point_surface_distance(point, duck1)
    dist2 = point_point_distance(point, duck1.evalpts)
    err += abs(dist - dist2)
print(err/ 1000)
```

In this code snippet, we set the sample size to be 100 in $u$ and $v$ direction, which means we sample 10000 points from the surface and store the approximation error to **err** variable. We execute the code 1000 times and it turns out the average error is very low, which is roughly $10^{-4}$. Thus we can compute the distance between points to approximate the distance between points and a surface.

# 5 Approach

## 5.1 Pipeline

### 5.1.1 An overview of pipelines

We have two major types of pipelines to illustrate our work on NURBS surfaces. One type of pipeline is supervised pipelines, follow the idea of the NURBS-diff paper [DBS+22], indicating the parameters of NURBS surfaces are known beforehand. We also tweak the original supervised pipeline in some respects with regard to loss type and optimizer. Another type adopts unsupervised learning methods. In unsupervised pipelines, unfortunately we lack any knowledge regarding the parameters, instead we could only utilize the point cloud sampled from NURBS or any objects. There are some variations to the pipelines as well. The goal of unsupervised pipelines is to find the ground truth parameters of NURBS surfaces by using unsupervised methods or at least enable the trained point cloud aligned with the ground truth point cloud.

Overall, these two types of pipelines are quite similar in general. The initial step is the loading of data. The data could be parameters or some point clouds. If an unsupervised method is adopted, the quality of the input point cloud plays a vital role and it will be explained in the following chapters. The quality of a point cloud refers to its density and distribution. In the next step, the point cloud to be trained is generated with initial parameters via forward propagation of the NURBS-Diff module. In addition, we also experiment with different sets of initial parameters. The training loop therefore starts to work. Afterwards, the trained point cloud is compared with the ground truth point cloud and the loss is computed. Following that, the loss is used to update the parameters. The training loop will be terminated when the loss is small enough or the number of iterations reaches the maximum. Finally, the trained output point cloud and control points are saved to files with OFF format separately. The parameters are also stored for later reconstruction.

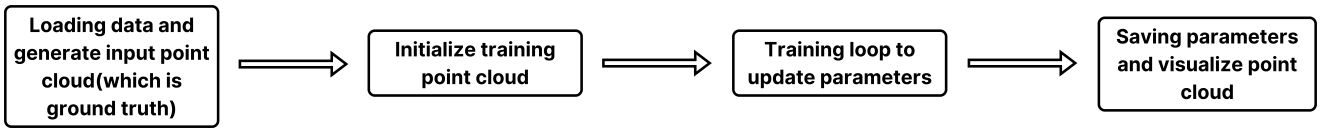



Figure 5.1: An overview of general pipelines

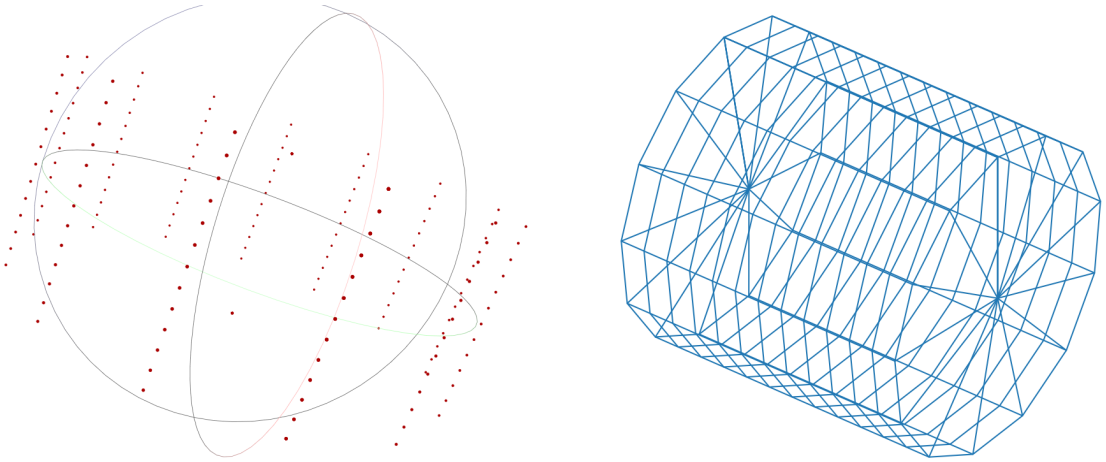
As illustrated in Figure 5.1, there are four main steps regarding all pipelines.

#### Loading data and generating ground truth point cloud

This step is similar to the original pipeline proposed in NURBS-Diff paper, but supervised and unsupervised pipelines cope with different data types. For supervised pipelines, the parameters of a NURBS surface should be loaded initially. Following, the forward propagation part of the NURBS-Diff module generates a corresponding point cloud by evaluating the surface at uniformly sampled 2D parametric coordinates. On the other hand, unsupervised pipelines will directly load a point cloud, which is sampled randomly, as the input. With either the ground truth parameters or some supersampling methods like Poisson disk sampling, we are able to obtain the input point cloud with high resolution. However, if we only get an input point cloud as the input, we are still able to increase the output resolution through the NURBS-Diff module.

### Generating initialized point cloud for training

The second step is the same for all approaches, which aims to initialize a NURBS-based point cloud for training. By applying the forward evaluation algorithm mentioned in the previous chapter, a point cloud with parametric sampling can be generated for training. To evaluate the influence of initial control points, we implement pipelines



(a) Initial cylinder-based control points (b) The wireframe of initial cylinder-based control points

Figure 5.2: Cylinder-based control points

using various initialization configurations. Besides random control polygons, the control points of basic primitives such as cylinders and sheets are implemented to define the training point cloud, as depicted in 5.2. Also, the initial control cages can be rotated with any angle, for the sake of simplicity, we set the initialized control points to be aligned along $x$, $y$, or $z$ axis.

### Training loop to update parameters

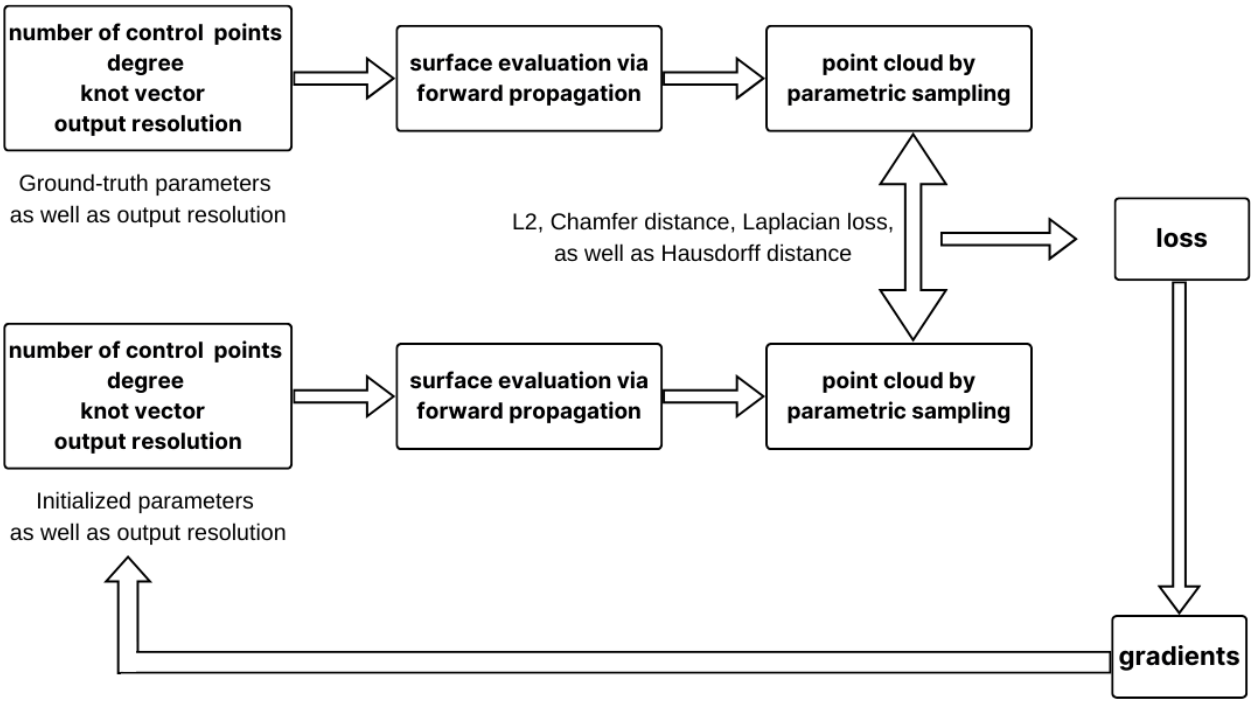
The third step is the training process. In most pipelines, the NURBS-Diff module aforementioned is utilized for generating predicted point clouds and computing gradients for parameter updates. Besides, we investigate different hyperparameters that may impact training process. For instance, we tweak the number of iterations, the loss function, and the predicted parameters.

Furthermore, in a specific unsupervised pipeline, we also discard NURBS-Diff module and modify the training loop to minimize the total distance from points in the input, to the training NURBS surface. As described in section 4.3, the method can be approximated by minimizing the distance from points to the training point cloud, given sufficient points are sampled from the surface.

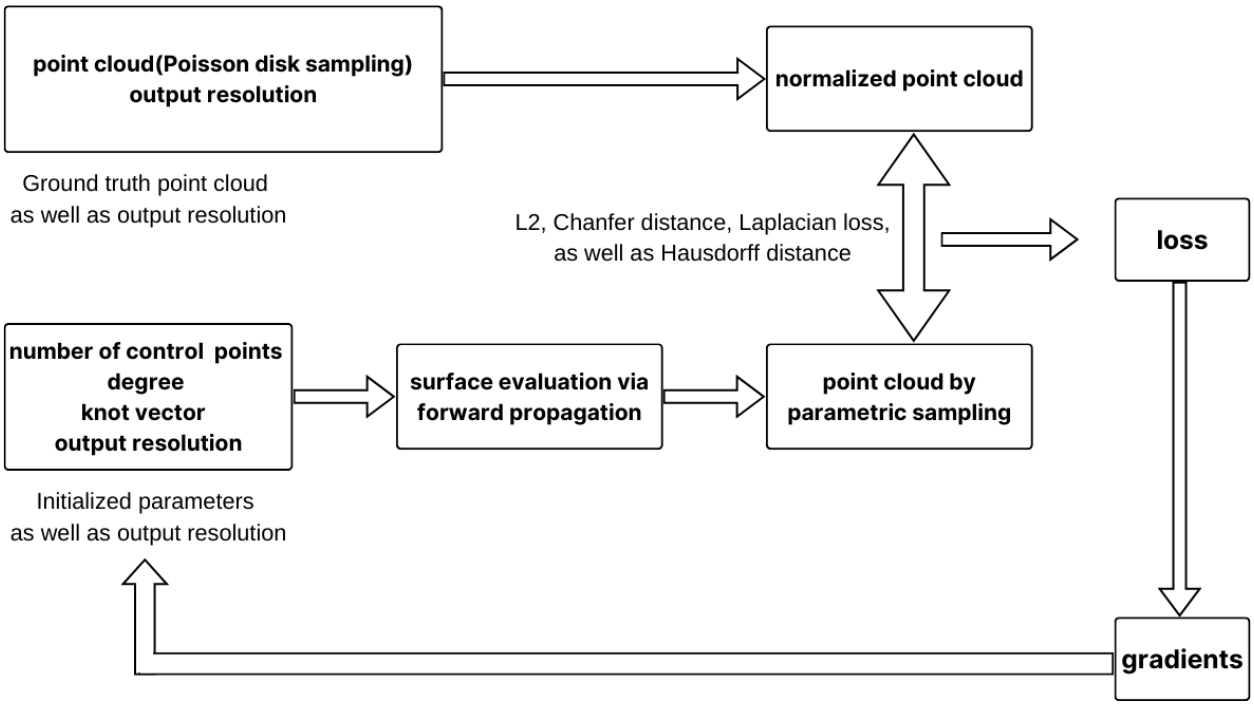
### Saving data and Plotting point clouds

After training for dedicated iterations or reaching the minimum loss, we save the trained point cloud to a file of OFF format. To reconstruct the NURBS surface, we also save the trained parameters accordingly. To be more specific, we plot these point clouds in 3D space by using **plot_wireframe**[1] function. The advantage of this function is to display the point cloud in an implicit order, which can be used to evaluate the quality of the reconstruction. The more ordering property it preserves, the better result it has. In this way, displaying the point cloud by using **plot_wireframe** makes sense. Besides, we also observe unordered point clouds in MeshLab.

[1] https://matplotlib.org/stable/gallery/mplot3d/wire3d.html




(a) Supervised pipeline




(b) Unsupervised pipeline

Figure 5.3: Supervised and unsupervised pipelines

### The choice of loss functions

For supervised pipelines, we not only use L2 loss, but also leverage the use of Chamfer distance as well as other losses. L2 loss is suitable for the supervised task since two point clouds in comparison are generated by the froward propagation part of the NURBS-Diff module, owning the same parametric order and the only difference lies in the parameters. As shown in Figure 5.3(a), supervised pipelines generate the target parametric point cloud and the predicted point cloud following the same process. Therefore, by comparing points pairwise, L2 loss provides an efficient way to evaluate how well the predicted point cloud matches the ground truth one. Based on our knowledge regarding point cloud comparison, we also leverage Chamfer distance, Hausdorff distance, and Laplacian matching loss to align point clouds. Chamfer distance and Hausdorff distance are two commonly used metrics to evaluate the alignment of point clouds. Laplacian matching loss, as introduced before, is a function which applies Laplacian operators to control point clouds and computes the difference in order to ensure control points are tight to the training surface instead of roaming randomly in 3D space.

While for unsupervised learning tasks, we apply the same loss functions as supervised methods in various pipelines. However, we mainly focus on Chamfer distance and Laplacian matching loss rather than L2 loss here because it is impossible to directly obtain two point clouds ordered in the same way. Under this circumstance, L2 loss works quite bad. Chamfer distance, instead, ignores the ordering in point clouds should perform better. The way to calculate Laplacian matching loss should be altered as well. Without ground-truth control points, Laplacian loss function, instead, should minimize itself rather than the difference.

We discuss the use of Laplacian loss a bit further. In the paper [SLK+20], the authors sample ground-truth B-spline patches uniformly by parametric sampling first, in the following, they measure the surface Laplacian capturing its second-order derivatives. Continuing the workflow, this method is also applied to predicted B-spline patches. Following that, they compare the Laplacians of ground-truth points and predicted points to minimize the discrepancy between their derivatives. The implementation of Laplacian loss in Python is shown in Listing 5.1.

**Listing 5.1: Code snippet for Laplacian loss**

```
    def laplacian_loss_splinenet(output, gt, dist_type="l2"):
        """
        Computes the laplacian of the input and output grid and defines
        regression loss.
        :param output: predicted control points grid. Makes sure the orientation/
        permutation of this output grid matches with the ground truth orientation.
        This is done by finding the least cost orientation during training.
        :param gt: gt control points grid.
        """
        batch_size, grid_size, grid_size, input_channels = gt.shape
        filter = ([[[0.0, 0.25, 0.0], [0.25, -1.0, 0.25], [0.0, 0.25, 0.0]],
                   [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
                   [[0, 0, 0], [0, 0, 0], [0, 0, 0]]])
        filter = np.stack([filter, np.roll(filter, 1, 0), np.roll(filter, 2, 0)])

        filter = -np.array(filter, dtype=np.float32)
        filter = Variable(torch.from_numpy(filter)).cuda()

        laplacian_output = F.conv2d(output.permute(0, 3, 1, 2), filter, padding=1)
        laplacian_input = F.conv2d(gt.permute(0, 3, 1, 2), filter, padding=1)
        if dist_type == "l2":
            dist = (laplacian_output - laplacian_input) ** 2
        elif dist_type == "l1":
            dist = torch.abs(laplacian_output - laplacian_input)
        dist = torch.sum(dist, 1)
        dist = torch.mean(dist)
        return dist
```

This loss function focuses on the regression of control points. As unsupervised pipelines lack ground-truth control point information, it is not feasible to regress control points. The paper [DBS+22] proposes a novel way to compute Laplacian loss for point cloud reconstruction. Instead of computing loss over control points, they

calculate it by minimizing the Laplacian of predicted point cloud directly. Since the code is not available for us, so we instead approximate it as follows:

**Listing 5.2: Code snippet for unsupervised Laplacian loss**

```
    def laplacian_loss_unsupervised(output, dist_type="l2"):
        filter = ([[[0.0, 0.25, 0.0], [0.25, -1.0, 0.25], [0.0, 0.25, 0.0]],
                   [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
                   [[0, 0, 0], [0, 0, 0], [0, 0, 0]]])
        filter = np.stack([filter, np.roll(filter, 1, 0), np.roll(filter, 2, 0)])
        filter = -np.array(filter, dtype=np.float32)
        filter = Variable(torch.from_numpy(filter)).cuda()
        laplacian_output = F.conv2d(output.permute(0, 3, 1, 2), filter, padding=1)
        if dist_type == "l2":
            dist = torch.sum((laplacian_output) ** 2, (1,2,3))
        elif dist_type == "l1":
            dist = torch.abs(torch.sum(laplacian_output.mean(),1))
        dist = torch.mean(dist)
        return dist
```

As the code shows, we remove everything regarding the ground truth, and minimize the Laplacian itself. Specifically, we hypothesize minimizing the second derivative of point position can reduce noise and enable generated wireframes with higher continuity or in partial order. As an extension to this idea, we also explore directly applying the Laplacian operator to predicted control points. By smoothing the control polygon rather than the final surface, we aim to enforce smoothness earlier in the generative process. Since control points play a vital role in deciding the position of output points, operating on control points could probably also work.

To clearly outline our pipelines, we illustrate most of them in Table 5.1. This table provides an overview of their abbreviation, descriptive details, and constituent parts. However, there are two separate pipelines which are quite different from the following, namely pipeline **UM** and **UP2P**, which we will explain later.

### 5.1.2 Ablation study: Supervised pipelines using NURBS-Diff module

The supervised pipelines basically follow the original surface fitting pipeline stated in the NURBS-Diff paper [DBS+22]. However, we also tweak the original pipeline in some respects. An overview of the pipelines is illustrated in Figure 5.3(a).

#### Pipeline SB

This baseline pipeline is absolutely the same as the one presented in the paper [DBS+22]. We follow every step in the original codebase[2], which aims to reconstruct the duck model using the NURBS-Diff module. Since it is a supervised method, thus the parameters of the duck model can be loaded from weights and ctrlpts files directly. Moving to the next stage, the ground truth point cloud is generated by using forward propagation part of the NURBS-Diff module. In the next step, there is the initialization of parameters for training. Following that, the training parameters are initialized by setting the weights and knot vectors to all-one vectors and randomly initializing the control points.

With the initialized fitting parameters, the trained point cloud is generated and compared to the ground truth point cloud using L2 loss, which is backpropagated to update the parameters. During the training process, the point cloud for training is constantly updated by trained parameters.

The training loop iterates until either the loss falls below a threshold or the maximum number of iterations is reached. Finally, the parameters are stored for reconstruction and a NURBS point cloud is generated.

Several new pipelines were created by implementing variations on the original pipeline. Including pipeline **SB**, the coefficients of these loss functions are fine-tuned to optimize the performance. The parameters generated from the pipeline achieving the optimal supervised performance are further used to restructure the surface model using the geomdl library.

[2]https://github.com/idealab-isu/NURBSDiff/blob/master/examples/DuckyNURBSSurfaceFitting.py

Table 5.1: An overview of most involved pipelines

| Pipeline Name | Brief description of different pipelines | Supervised | Loss Type | | | |
|---|---|---|---|---|---|---|
| | | | l2 | Chamfer | Hausdorff | Laplacian |
| SB | Original supervised pipeline in the NURBS-Diff paper, as the baseline for supervised pipelines | ✓ | ✓ | | | |
| SC | Supervised pipeline only training control points, using L2 loss | ✓ | ✓ | | | |
| SOC | Supervised pipeline only training control points, using L2 loss and Chamfer distance | ✓ | ✓ | ✓ | | |
| SOCH | Supervised pipeline only training control points, with L2 loss , Chamfer and Hausdorff distance | ✓ | ✓ | ✓ | ✓ | |
| SOCHL | Supervised pipeline only training control points, with L2 loss, Chamfer and Hausdorff distance and Laplacian loss | ✓ | ✓ | ✓ | ✓ | ✓ |
| UB | Unsupervised pipeline adapted from SB, as the baseline for unsupervised pipelines | | ✓ | | | |
| UC | Unsupervised pipeline using Chamfer distance, working with point clouds of same sizes | | | ✓ | | |
| UCD | Unsupervised pipeline using Chamfer distance, working with point clouds of different sizes | | | ✓ | | |
| UCDC | Unsupervised pipeline using Chamfer distance, working with point clouds of different sizes, along with the preset initialized control points | | | ✓ | | |
| UCDL | Unsupervised pipeline using an assembly of Laplacian loss and Chamfer distance working with point clouds of different sizes, along with preset points | | | ✓ | | ✓ |
| UCDHL | Unsupervised pipeline using an assembly of Laplacian loss, Chamfer distance and Hausdorff distance working with point clouds of different sizes, along with preset control points | | | ✓ | ✓ | ✓ |
| UCDH | Unsupervised pipeline using an assembly of Chamfer distance and Hausdorff distance working with point clouds of different sizes, along with a preset control points | | | ✓ | ✓ | |

#### Pipeline SC

When analyzing pipeline **SB**, we observe the training of knot vectors and weights does not yield satisfying results since the gradients of these parameters are trivial to be ignored. Therefore, we decide to only train the control points in pipeline **SC** and fix weights and knot vectors to be the ground truth.

#### Pipeline SOC

Additionally, we also experiment with the workflow of the original pipeline except that we use other loss functions. In pipeline **SOC**, a combination of L2 and Chamfer loss is adopted for training. Chamfer distance is used for further decreasing the discrepancy between points clouds.

#### Pipeline SOCH

While in pipeline **SOCH**, L2, Chamfer, and Hausdorff loss are utilized for the training loop. Hausdorff loss here works similar to Chamfer distance, aiming to align point clouds better.

#### Pipeline SOCHL

Finally, we conduct the experiment with an assembly of L2, Chamfer, Hausdorff, and Laplacian loss in pipeline **SOCHL**. Laplacian loss is utilized to enable edges connected between points much smoother.

### 5.1.3 Ablation study: Unsupervised pipelines using NURBS-Diff module

The unsupervised pipelines are much more significant and complicated than the supervised pipelines, since in this case, the module may not work when the input point cloud is unordered. So we also explore multiple pipelines with variations by adjusting types of loss function, normalization of input point cloud, and some tweaks in the training loop. The main steps for general unsupervised pipelines are shown in Figure 5.3(b).

#### Pipeline UB

The baseline unsupervised pipeline **UB** migrates from the original supervised pipeline **SB** by keeping all steps the same except that the parameters are unknown in this case. Unlike supervised pipelines, we have no access to parametric sampled point clouds. Instead, we can only obtain the point cloud from vertices stored in OFF format files or some supersampling methods like Poisson disk sampling. We are capable of obtaining any resolution of the input point cloud, however, these sampling ways have a critical drawback, which is, the sampled point cloud under this circumstance is unlikely to retrieve the intrinsic parametric information contained in the underlying surface. This scenario is very common in the real world since point clouds and mesh-based objects are widely used rather than underlying parameters.

Therefore, although we can obtain the ground truth point cloud with high resolution, it is unlikely to reveal any pattern of NURBS surfaces. In other words, point clouds sampled randomly from NURBS or other objects are utilized as the input point cloud for training. Next, the NURBS-Diff module generates the initialized point cloud and trains it by minimizing the discrepancy with the input point cloud. Considering the unsupervised settings, the quality of the input point cloud plays a vital role in the training process since the only difference between this pipeline and the original supervised pipeline lies in the input method while the remaining steps stay the same. If we could generate a point cloud closely resembling the pattern of NURBS surfaces, pipeline **UB** and **SB** should follow the same way. However, as discussed above, the point cloud is always randomly sampled pragmatically. Based on this limitation, we should tweak the pipelines from other aspects illustrated in the following.

#### Pipeline UC

The order in the input point cloud can not be guaranteed in any unsupervised pipeline. We have also verified it by examining a wireframe object which rendered from an unordered point cloud shown in Figure 3.4. The point

cloud clearly lacks any defined ordering of points in 3D space. Therefore, it is necessary to adjust the loss function because we can no longer access the mapping relationship between paramedic coordinates and 3D space in the context of unsupervised pipelines. Instead, we change capable to change loss type from L2 to Chamfer distance, which focuses on the alignment of point clouds and ignores any order in points. This modification using only Chamfer loss is to optimize unordered point clouds in the absence of ground-truth parameters during the fitting process.

### Pipeline UCD

Chamfer distance defined in Equation 2.7, allows for point clouds of different sizes to be compared, therefore we can compare the target point cloud with a scaled training point cloud, here scaling means increasing the number of points. Training point clouds with more points should reduce the loss, since they contain more geometric information, which helps better align with the target point cloud. Based on this, we propose another unsupervised pipeline called **UCD** that utilizes Chamfer distance as the loss function while the size of the training point cloud is set to be $100 \times 100$, which is much larger than the input (the size of the input point cloud is $50 \times 50$ average). This scaling approach should dramatically reduce the loss during alignment compared to using point clouds of equal size.

### Pipeline UCDC

Pipeline **UCD** boosts the performance of trained point clouds. However, there are several notable issues observed with the output point cloud. The main issue is the trained control points seem rather random in comparison to the original supervised pipeline. The control polygon in pipeline **UCD** deviates far away from the ground truth control points shown in supervised learning methods. Another issue is the trained point cloud exhibits too much noise, which is also caused by the irregularity of the control point grid. Related measures are taken into account

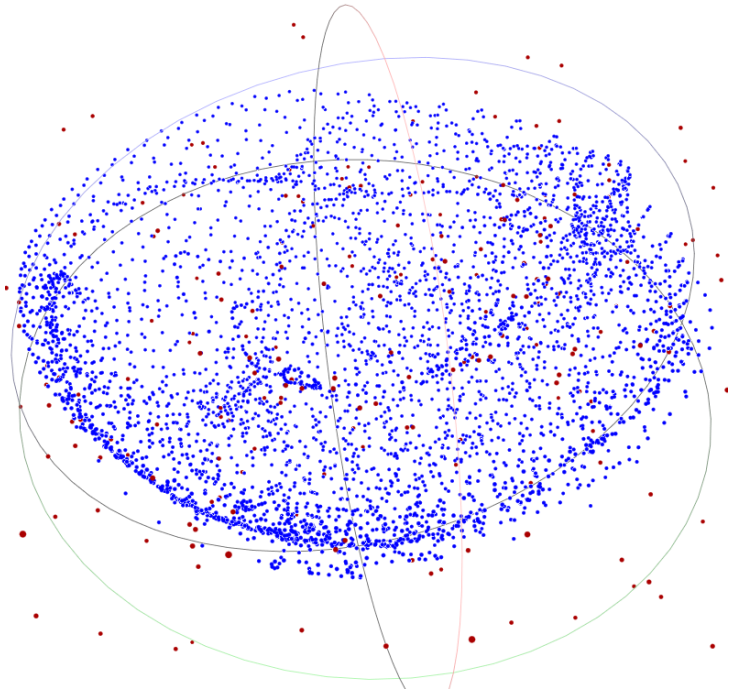


Figure 5.4: A half sphere containing noise inside, with an initial cylinder cage

to solve these issues. We believe randomized initialization of control points causes the problem, thus we decide to use control points from known objects as initial parameters to cover target point cloud. We experiment with the control points of a cylinder first, which is tested to work well for closed surfaces. However, it does not

dramatically improve results for open surfaces. As observed in Figure 5.4, all control points should be above the half sphere, but the initialization forces some below the target. These control points are not able to position themselves accurately eventually.

So to address this problem, the open control polygon of sheets is used as the initial control point set for open surfaces. Sheet control points are also tested on corresponding surfaces, which works better compared to the cylinder control polygon.

Generally, in pipeline **UCDC**, we follow the same workflow as pipeline **UCD** except that we use control points of basic NURBS surfaces as initial parameters for closed and open surfaces respectively.

#### Pipeline UCDL

The pipeline **UCDC** achieves a lower loss compared with pipelines aforementioned, however, the trained point cloud still exhibits noise that requires further refinement. To get rid of the noise, we also explore incorporating Laplacian matching loss into the loss function to train the parameters.

The Laplacian loss helps minimize local geometric distortion by encouraging points to remain close to their neighbors, which is a feasible option to mitigate the effect of the noise by keeping points close to each other. In supervised learning methods, the Laplacian loss is computed by the L2 norm between the Laplacian operators to two control points clouds (target and predicted one). However, when it comes to unsupervised learning, ground-truth parameters are not able to be obtained. In this way, in order to compute Laplacian loss, we directly minimize the L2 norm of the Laplacian operators to predicted output point cloud, which is also stated in the NURBS-diff paper [DBS+22]. Since control points control the position of output points, therefore, we can leverage this property by applying Laplacian loss to the predicted control points directly, afterwards, the gradient is computed to update control points, which updates the position of output point cloud accordingly.

When combined with Chamfer loss, the assembly of the multi-loss function aims to output cleaner point clouds with mitigated noise while keeping high fidelity to the target point cloud.

#### Pipeline UCDHL

Based on pipeline **UCDL**, we experiment with the combination of Chamfer, Laplacian, and Hausdorff loss in pipeline **UCDHL** since Hausdorff distance is another metric like Chamfer distance to evaluate the alignment of point clouds. We would like to investigate whether Hausdorff distance could further improve the performance.

#### Pipeline UCDH

Also, we evaluate the performance of a combination of Chamfer and Hausdorff loss in our pipeline **UCDH**. This pipeline discards Laplacian loss, aiming to investigate if Laplacian loss truly improves the final result.

### 5.1.4 Two alternative unsupervised pipelines

A key challenge in any unsupervised method is the input point cloud is unstructured which is quite different from supervised pipelines. In supervised pipelines, there is an inherent correspondence between input and training point clouds. Therefore, we seek to order the input point cloud like parametric sampling, or we can adopt another way which discard the NURBS-Diff module and turns to the point-to-surface approach.

In this subsection, we illustrate another two sophisticated unsupervised pipelines in comparison to previous pipelines. Pipeline **UM** (an unsupervised pipeline for matching two point clouds) aims to impose an ordered correspondence between input point cloud and generated NURBS-based point cloud. While pipeline **UP2P** (an unsupervised pipeline computing point to point cloud distance) fits the target directly to the parametric NURBS surfaces produced during training. The pipeline works by minimizing the total distance from each target point to the nearest point on the parametric training surface.

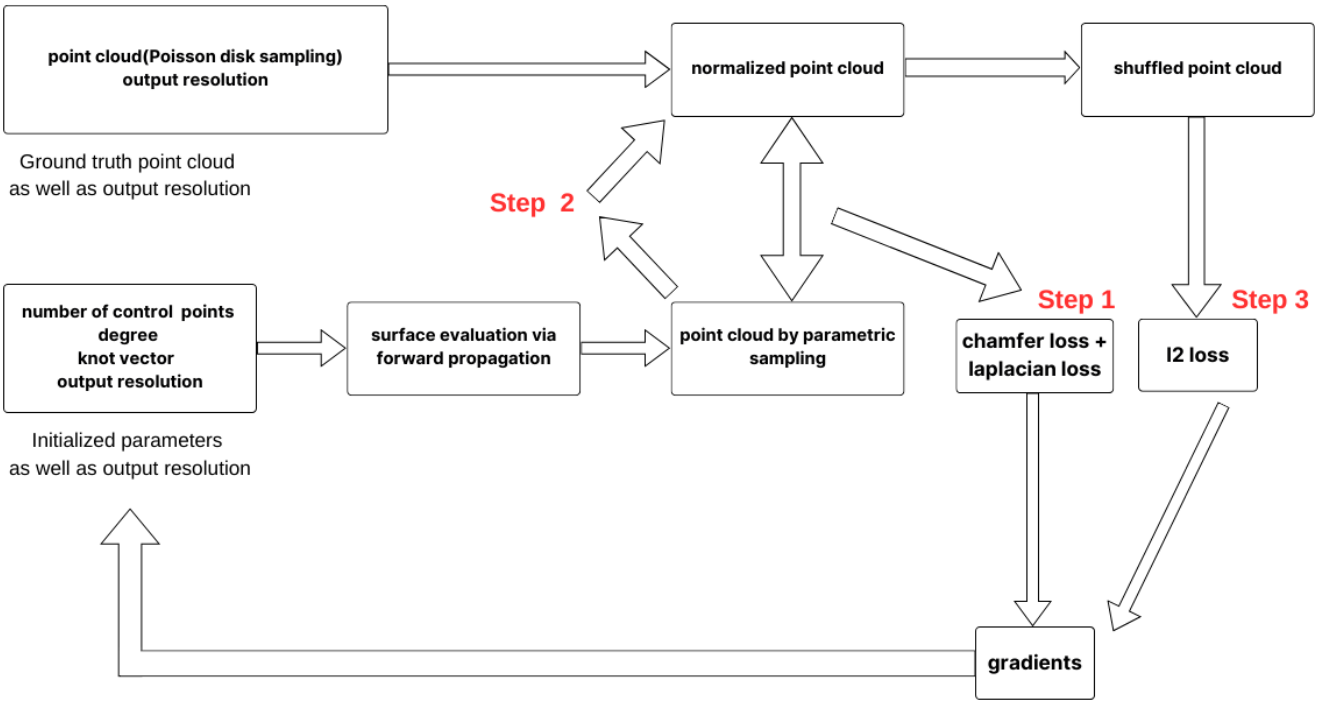



Figure 5.5: Pipeline UM

### Pipeline UM

Considering the unordered input point cloud, in pipeline **UM**, we aim to enable an order property for the input. First, similar to pipeline **UCDL**, we compute Chamfer loss to decrease the discrepancy between two point clouds. Our hypothesis is after the first step, the input and output point clouds should be in correspondence since these point clouds are aligned, thus the input point cloud should also be in order as the output does. Therefore, we utilize **minimize** method from **scipy.optimize** module, which is illustrated in Listing 4.1, to minimize one-to-one mapping distance between two point clouds. In other words, we enable each point in the input point cloud to have a corresponding point in the generated input cloud, while the total distance achieves the minimum. According to the imposed mapping, we subsequently shuffle the order of points in the input point cloud. Afterwards, in the final step, we compute L2 loss for further matching two "ordered" point clouds.

So to sum up, pipeline **UM** minimizes Chamfer loss in step one. In the subsequent step, the input point cloud gains order property by pairwise correspondence. Finally, L2 loss is calculated between two points, and the gradient is backpropagated to update corresponding parameters. These steps are considered as a single iteration. We always execute it for several iterations.

### Pipeline UP2P

Furthermore, we discard the NURBS-Diff module to do the backpropagation anymore since it only works well with supervised input (point cloud with the spiral pattern) rather than uniformly sampled input (Poisson disk sampling). Instead, in pipeline **UP2P**, we minimize the distance between the input point cloud and the predicted NURBS surface to enable the training NURBS surface to constantly approach the ground truth point cloud.

With the help of SciPy package, we can implement the code like the program shown in Listing 4.1, which is helpful for us to compute point-to-surface distance. From here, we can obtain the total loss, however, the crucial thing is to backpropagate the loss through all network layers for the update. As geomdl states, the package is based on pure Python, hence it does not support any Pytorch operation, which further indicates the loss we obtained here is not a leaf node[3]. If we directly apply it in the training process, the loss will not be updated.

[3]https://pytorch.org/docs/stable/generated/torch.Tensor.is_leaf.html

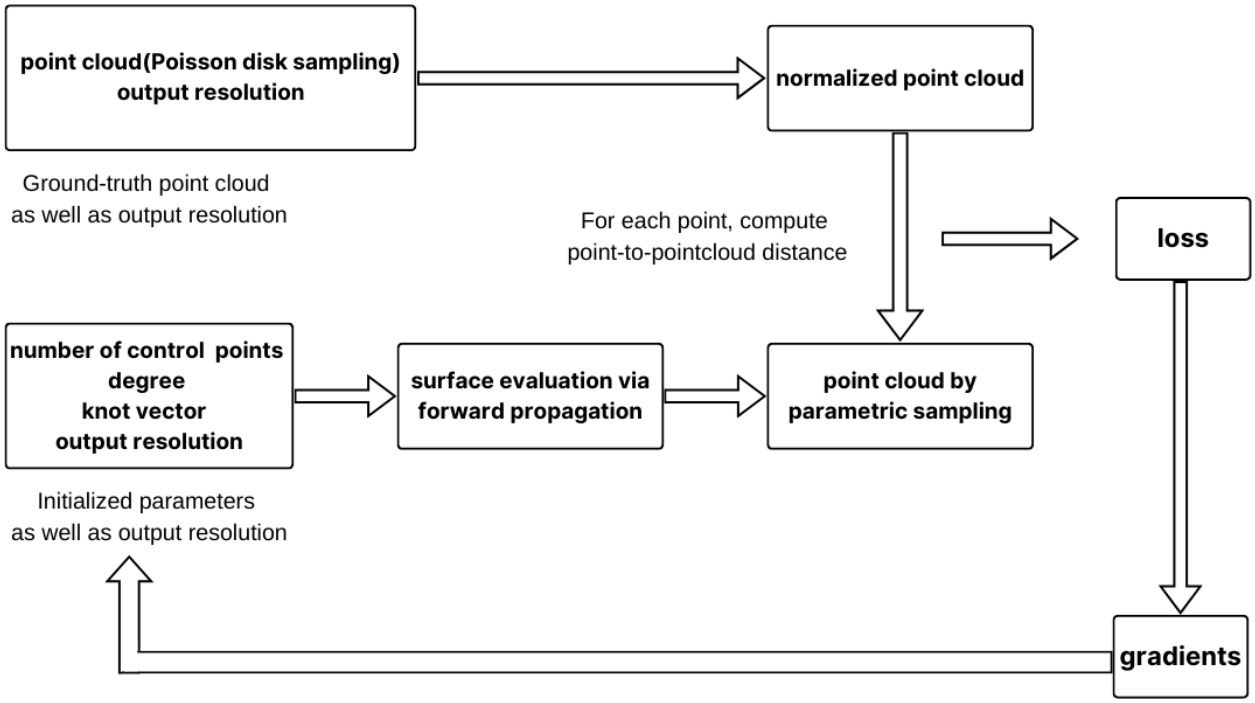



Figure 5.6: Pipeline UP2P

There are two ways to tackle this issue. On one hand, it is possible to implement torch-version geomdl, especially refactoring forward propagation methods like **evaluate_single**. On the other hand, there is a simpler way to approximate the process. As discussed in the paper [PGK02], the distance can be approximated from the domain of point-to-surface to the domain of point-to-point. Also, we verify this algorithm in section 4.3. Therefore, we do not need to compute any surface, instead, the only requirement is to scale the size of the training point cloud. The generated point cloud should be several times larger than the ground truth point cloud. Restricted by machine memory, we are able to generate a point cloud based on initial parameters with size ranging from $100 \times 100$ to $150 \times 150$. The whole process is explained in Figure 5.6.

In short, we experiment with several pipelines with variation under unsupervised learning settings. Under this circumstance, since the quality of the input point cloud is somehow ”bad” compared to supervised pipelines, we tweak the pipelines by changing training loops as well as loss types. We also come up with a point-to-point solution without the NURBS-Diff module.

## 5.2 Visualization tools

For visualization, we take advantage of the powerful PyQt5 library and visualize the pipelines above through it. According to Riverbank Computing[4], the author of PyQt5, PyQt5 is an extensive set of Python bindings of Qt[5], which is another set of C++ libraries which support GUI applications. Through PyQt5, we can leverage Qt's cross-platform capabilities to quickly implement our pipeline into a GUI application.

In general, there are three key files which comprise our visualization implementation in PyQt5 as follows.

*supervised.py* contains all supervised pipelines. With **supervised_type** variable provided by *vis.py*, different pipelines can be executed to generate a point cloud. Similar to *supervised.py*, *unsupervised.py* contains various unsupervised pipelines. Different pipelines will be called by parameter **unsupervised_type** from *vis.py*.

*vis.py* is another important file which serves as the primary interface, controlling pipeline execution, parameterization, and visualization. In the following, we will explain these four steps in *vis.py* in order.

[4]https://www.riverbankcomputing.com/software/pyqt/
[5]https://www.qt.io/

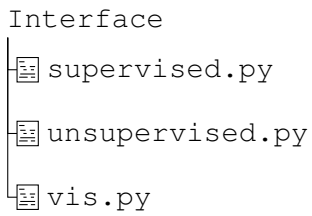



Figure 5.7: The structure of the interface

### Loading point cloud

In supervised learning pipelines, we need to load control points as well as weights. While in unsupervised learning pipelines, we need to load point clouds which are the ground truth sampled by Poisson disk sampling. We can load the parameter files using the following code.

**Listing 5.3: Code snippet for loading point cloud**

```
    ...
    self.load_button = QPushButton('Load Input Point Cloud')
    self.load_button.clicked.connect(self.load_point_cloud)
    def load_point_cloud(self):

        # Open a file dialog for loading the point cloud
        options = QFileDialog.Options()
        options |= QFileDialog.ReadOnly
        file_name, _ = QFileDialog.getOpenFileName(self, 'Open Point Cloud', '', 'Point
            Cloud Files (*.pcd);;NOFF Files (*.noff);;OFF Files (*.off);;All Files (*)',
             options=options)

        if file_name:
            self.remove_point_clouds()
            # Load the point cloud from file
            with open(file_name, 'r') as f:
                ...
```

In the code, we first instantiate a **QPushButton**, which is to trigger the dialog box. The dialog box should appear for file selection upon the click. Once a file is confirmed, the dialog box should be closed. Next, the file path is passed to variable **file_name**. In this way, we are able to parse the file content line by line.

### Switch functionality

**Listing 5.4: Code snippet for switching tabs**

```
    ...
    self.tab_widget = QTabWidget()
    self.tab_widget.addTab(unsuperivsed_central_widget, "Unsupervised")
    self.tab_widget.addTab(supervised_central_widget, "Supervised")
    self.tab_widget.addTab(geomdl_central_widget, "NURBS Generator")
    ...
```

We need to switch among several tabs for different approaches. For instance, the user may need to interactively toggle between a supervised pipeline and an unsupervised pipeline, which can be accomplished by **QTabWidget** and its **addTab** method. As shown in Listing 5.4, the code for switching can be implemented in this way.

By adding tabs to the main **QTabWidget**, we can easily switch among them with simple clicks.

### Setting parameters

**Listing 5.5: Code snippet for setting parameters**

```
    ...
    out_dim_v_layout = QHBoxLayout()
    out_dim_v_label = QLabel("Out Dim v: ")
    out_dim_v_layout.addWidget(out_dim_v_label)
    self.out_dim_v_spin_box_1 = QSpinBox()
    self.out_dim_v_spin_box_1.setMinimum(8)
    self.out_dim_v_spin_box_1.setMaximum(128)
    self.out_dim_v_spin_box_1.setMaximumSize(35, 20)
    self.out_dim_v_spin_box_1.setMinimumSize(35, 20)
    self.out_dim_v_spin_box_1.setValue(self.out_dim_v)
    out_dim_v_layout.addWidget(self.out_dim_v_spin_box_1)
    out_dim_v_layout.setContentsMargins(0, 15, 0, 30)
    spin_box_layout.addLayout(out_dim_v_layout)
    ...
```

This step is of great significance in the whole process. We need to dynamically configure pipeline parameters at runtime. By doing this, **vis.py** separates hyperparameters from the deep learning modules. With the help of **QSpinBox** and **QLabel**, the following code in Listing 5.5 is to build an interactive interface for parameterization.

In Listing 5.5, we first initialize a horizontal box layout for the output dimension. Afterwards, we label it using **QLabel**. Following that, we limit the range of adjustable values in a **QSpinBox** and set the value to be an initial default value. Finally, we append this layout to the main interface.

### Visualization

We leverage **QtInteractor** and its **add_mesh** method for the visualization of point clouds and meshes. The following code snippet demonstrates its usage.

**Listing 5.6: Code snippet for visualization**

```
    ...
    # Visualize point cloud
    self.input_point_cloud_widget = QtInteractor()
    self.input_point_cloud_widget.add_mesh(output_point_cloud, render_points_as_spheres=
        True, point_size=5, color='cyan', name='input_point_cloud_supervised')

    # Visualize mesh
    self.geomdl_output_widget = QtInteractor()
    self.geomdl_output_widget.add_mesh(mesh, color='lightgreen', name='geomdl_mesh')
    ...
```

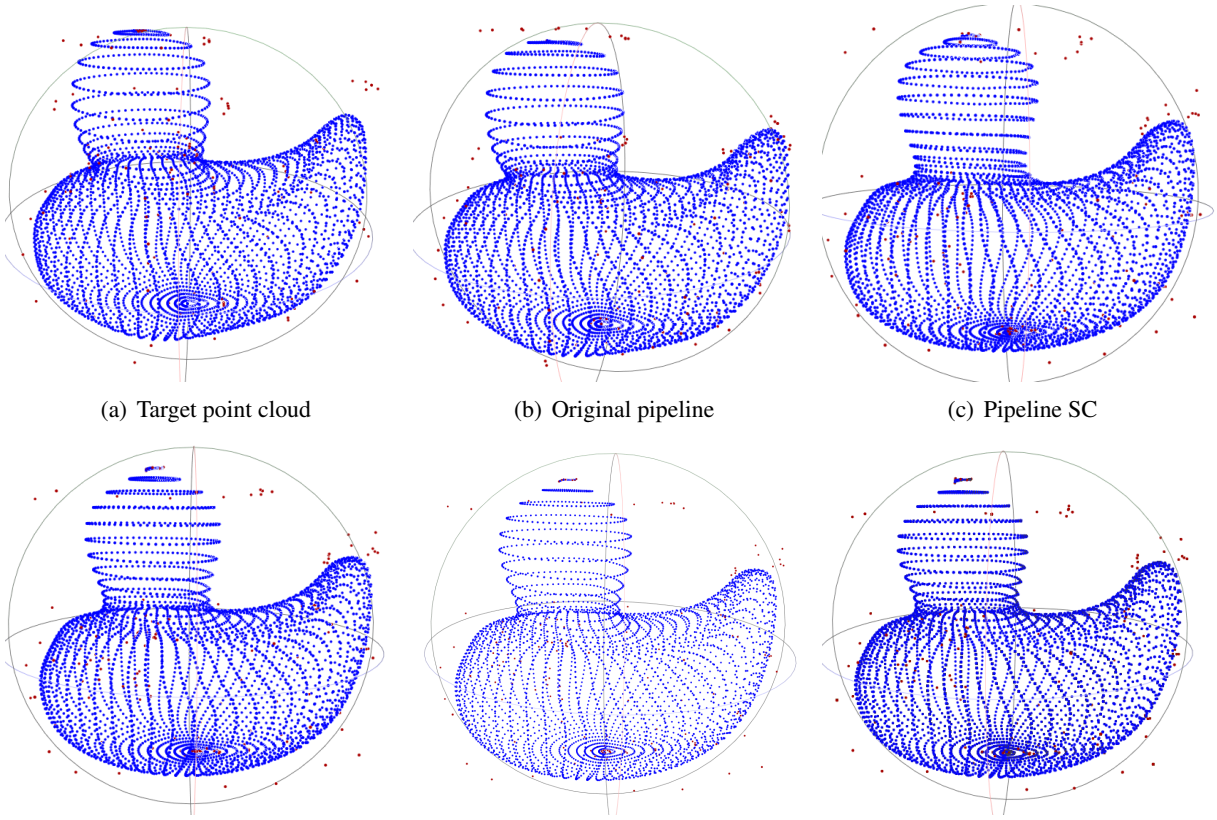
The visualization tool is now publicly available on github[6], which provides the necessary functionality for evaluation purposes.

[6]https://github.com/SyLi9527/NURBSDiff/tree/dev/examples/test/interface

# 6 Results

In this chapter, the results of pipelines are visualized and examined utilizing MeshLab and Matplotlib. First, we present the results of supervised pipelines, followed by the results of unsupervised pipelines. Due to the limited space of this master's thesis, we only show the results of the duck model and horse model here. Some results of other models can be found in Appendix A.

## 6.1 The result of supervised pipelines



(a) Target point cloud (b) Original pipeline (c) Pipeline SC

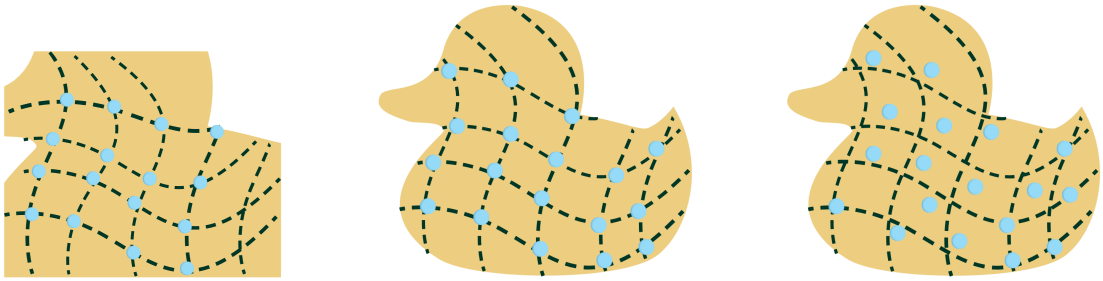
(d) Pipeline SOC (e) Pipeline SOCH (f) Pipeline SOCHL

Figure 6.1: Supervised pipelines of duck model. The number of control points in $u$ and $v$ direction are 14 and 13. The output resolution is $64 \times 64$.

We apply the supervised pipelines to the duck model. The results are displayed in Figure 6.1.

Since we have obtained duck model parameters, we are able to generate the ground truth point cloud with the parametric sampling pattern easily. As shown in Figure 6.1(a), the ground truth point cloud follows the pattern of NURBS surfaces. For pipelines shown in Figure 6.1(b), 6.1(c), 6.1(d), 6.1(e), and 6.1(f), trained point clouds are also aligned with the ground truth one. However, the trained control polygon, which is obtained by the original pipeline **SB**, resembles the ground truth polygon most closely. Also, the original pipeline achieves the minimum loss value using L2 loss. This is mainly because the input point is sampled uniformly in $u$ and $v$ directions from

the NURBS surface, while the initialized point cloud follows the same way, replicating the NURBS point cloud pattern.



(a) Parametric sampling (training) (b) Parametric sampling (ground truth) (c) Poisson disk sampling

Figure 6.2: Point clouds sampled in different ways

Therefore, it is efficient and accurate to align these point clouds by using L2 loss since the only goal is to find the one-to-one correspondences given two identically patterned point clouds. Figure 6.2 also illustrates the comparison. Since two point clouds in the original pipeline are sampled in the same order, we can directly measure them using L2 loss to enable loss minimization through parameter updates.

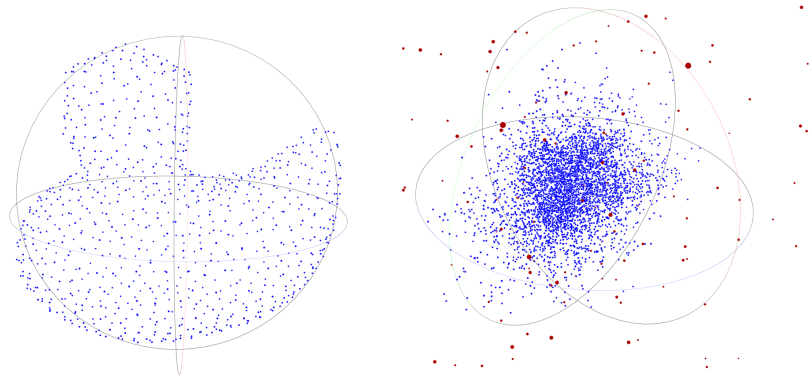
## 6.2 The result of unsupervised pipelines

Table 6.1: The loss table of the duck model ($15 \times 15$ control points)

| N. control pts | 15 x 15 | Model | duck | | | |
|---|---|---|---|---|---|---|
| Resolution Input | 30 x 30 | | | | | |
| Pipeline | Time Per Epoch | Loss | Description | Laplacian | Chamfer | Hausdorff |
| UB | 0.052 | 0.044 | L2 | | | |
| UCD | 0.044 | 0.00053 | CD | 0.000 | 0.00053 | 0.00 |
| UC | 0.046 | 0.00074 | CD (same size) | 0.000 | 0.00074 | 0.000 |
| UCDC | 0.043 | 0.00043 | CD (cages) | 0.000 | 0.00043 | 0.000 |
| UCDH | 0.066 | 0.00194 | CD + 0.01 * HD | 0.000 | 0.00067 | 0.127 |
| UCDL | 0.058 | 0.00057 | CD + 0.001 * LD | 0.159 | 0.00041 | 0.000 |
| UCDHL | 0.061 | 0.00203 | CD + 0.001 * LD + 0.01 * HD | 0.218 | 0.00057 | 0.124 |

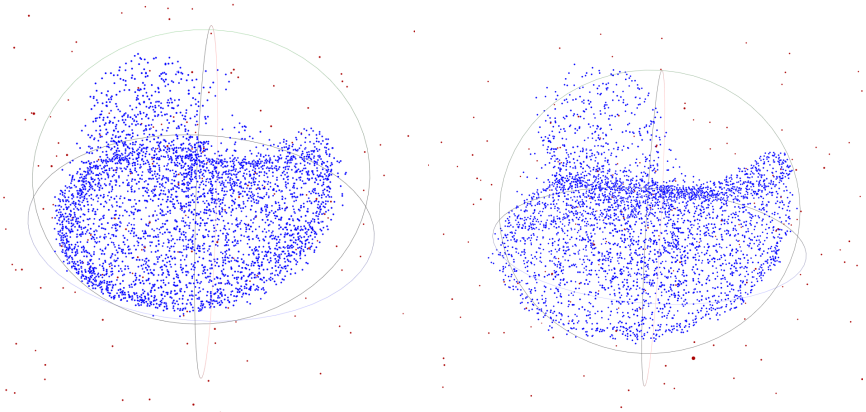
Table 6.2: The loss table of the horse model ($20 \times 20$ control points)

| N. control pts | 20 x 20 | Model | horse | | | |
|---|---|---|---|---|---|---|
| Resolution Input | 50 x 50 | | | | | |
| Pipeline | Time Per Epoch | Loss | Description | Laplacian | Chamfer | Hausdorff |
| UB | 0.054 | 0.129 | L2 | | | |
| UCD | 0.045 | 0.00087 | CD | 0.000 | 0.00087 | 0.00 |
| UC | 0.042 | 0.00091 | CD (same size) | 0.000 | 0.00091 | 0.000 |
| UCDC | 0.046 | 0.00069 | CD (cages) | 0.000 | 0.00069 | 0.000 |
| UCDH | 0.077 | 0.00305 | CD + 0.01 * HD | 0.000 | 0.00073 | 0.232 |
| UCDL | 0.070 | 0.00118 | CD + 0.001 * LD | 0.474 | 0.00071 | 0.000 |
| UCDHL | 0.072 | 0.00289 | CD + 0.001 * LD + 0.01 * HD | 0.59 | 0.00071 | 0.159 |



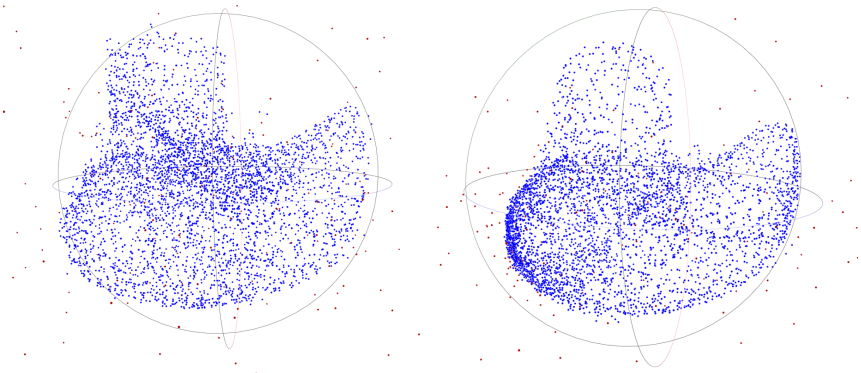
(a) Target point cloud

(b) Pipeline UB



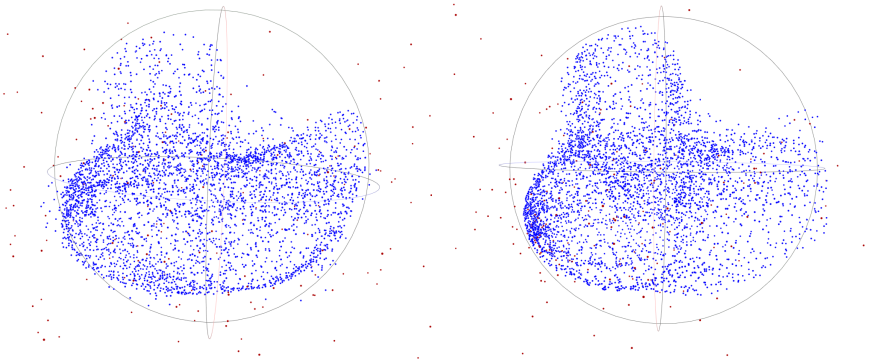
(c) Pipeline UC

(d) Pipeline UCD



(e) Pipeline UCDC

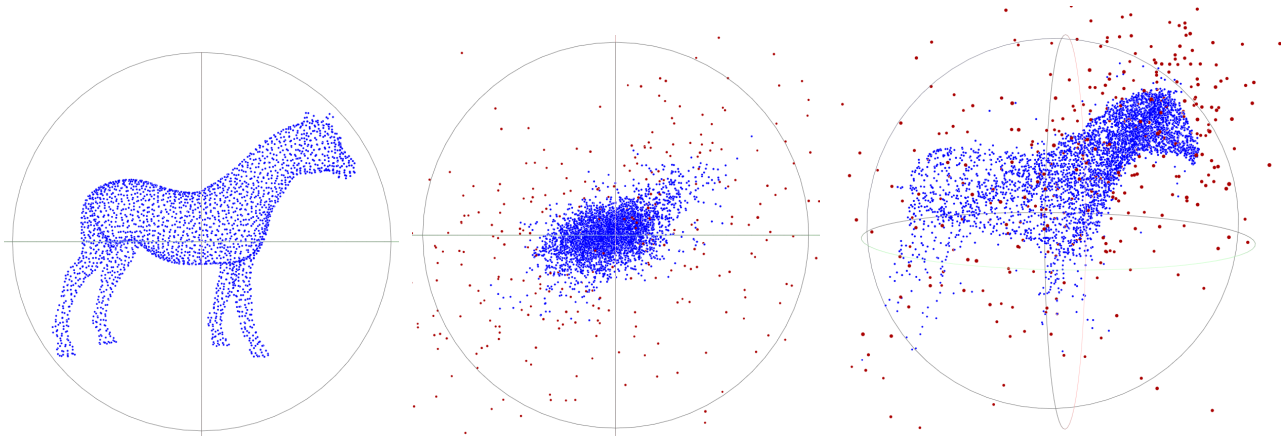
(f) Pipeline UCDL



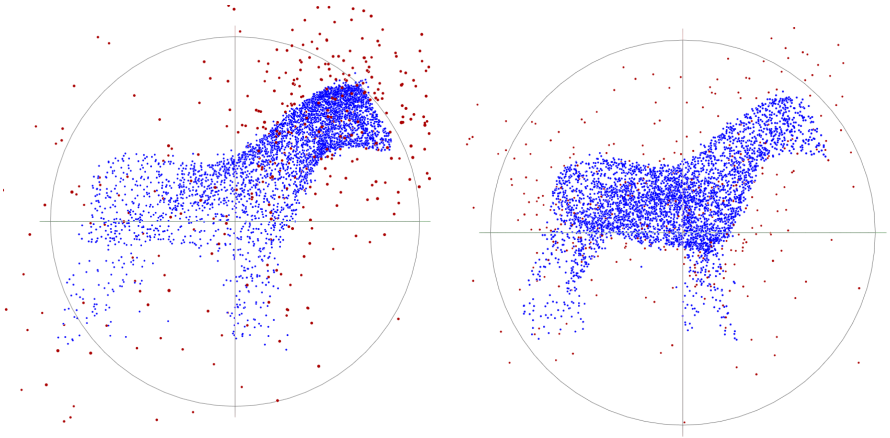
(g) Pipeline UCDH

(h) Pipeline UCDHL

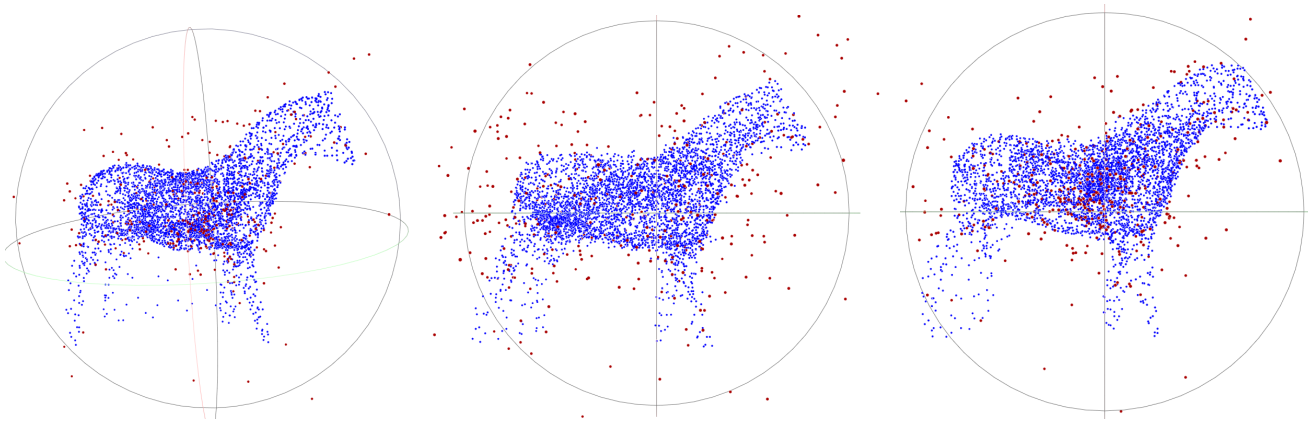
Figure 6.3: Unsupervised pipelines of duck model. The number of control points in $u$ and $v$ direction are set to 15 and 15, while the resolution of input point cloud is $30 \times 30$. The output resolution is $64 \times 64$. Red points are predicted control points while the blue ones are predicted output.



(a) Target point cloud (b) Pipeline UB (c) Pipeline UC



(d) Pipeline UCD (e) Pipeline UCDC



(f) Pipeline UCDL (g) Pipeline UCDH (h) Pipeline UCDHL

Figure 6.4: Unsupervised pipelines of the horse model. The number of control points in $u$ and $v$ directions are set to 20 and 20, while the resolution of input point cloud is $50 \times 50$. The output resolution is $64 \times 64$. Red points are predicted control points while the blue ones are predicted output.

We experiment with the unsupervised pipelines on the duck and horse model. The results are presented in Figure 6.3 and Figure 6.4 respectively. Since the pipelines in Figure 6.3 and 6.4 follow the same logic, we only analyze the duck model results here.

Figure 6.3(a), depicts the target point cloud sampled from the duck model using Poisson disk sampling, lacking any discernible pattern unlike the pattern shown in Figure 6.1(a). Although the duck model is a NURBS object, the target point cloud is not sampled uniformly in $u$ and $v$ direction, biasing from the supervised case. However, the training point cloud, computed by forward propagation of the NURBS-Diff module, follows the parametric sampling way. Therefore, direct l2 loss alignment gets an unsatisfying result, as shown in Figure 6.3(b). Accordingly, the total loss is 0.044, which is the highest among unsupervised pipelines shown in Table 6.1. In this case, we adopt other loss functions for the alignment task.

So in pipeline **UC**, we utilize Chamfer distance as the loss function. As shown in Figure 6.3(c), the trained point cloud is aligned with the target one. The loss of this pipeline greatly decreased to 0.00074 compared to pipeline **UB**. However, noise remains in the output. Besides, the control polygon is significantly far away from the ground truth.

To solve this issue, we experiment with pipeline **UCD** which aims to remove the noise by increasing the size of training point cloud in order to align better with the target point cloud. As shown in Figure 6.3(d), pipeline **UCD** largely mitigates noise by increasing training set size, achieving better alignment. Besides, the loss listed in Table 6.1 is 0.00053, which is lower than pipeline **UC**. However, the drawback of this pipeline is the output lacks the sharp corner on the duck's head.

We believe this is because we do not regulate the control points previously. Therefore, in pipeline **UCDC**, we refine initialized control points. As shown in Figure 6.3(e), the output is closely aligned with the target and it almost fully retrieves the shape of the duck head as well as its body. Moreover, the computed loss achieves the minimum among all unsupervised pipelines. However, we find control polygons remain inaccurate. As can be observed, the control points are not like the ground-truth one, which encapsulates the duck model tightly.

Under this circumstance, the Laplacian matching loss, which is referred to in ParSeNet [SLK$^+$20] and NURBS-Diff [DBS$^+$22], is utilized in order to enforce control points close to each other. Figure 6.3(f) shows pipeline **UCDL** produces tighter alignment and control polygon coherence to the NURBS structure. The position of control points is similar to the original one in Figure 6.1(a) except that the distribution of the control points is different. The loss of this pipeline is close to pipeline **UCD**, but output point cloud achieves a better alignment.

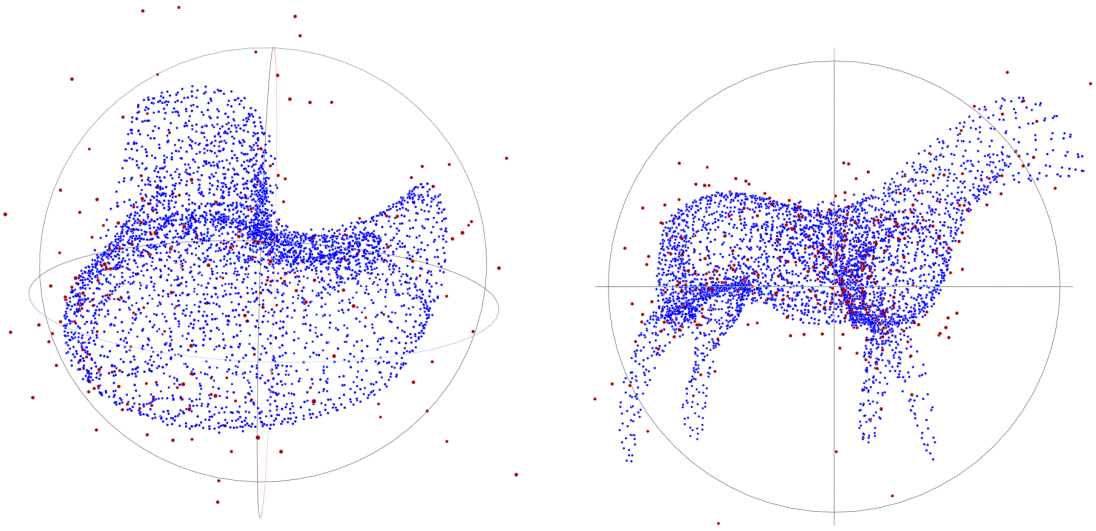
We also experiment with a combination of Laplacian, Hausdorff, and Chamfer loss in pipeline **UCDHL**. As shown in Figure 6.3(h), the trained point cloud does not have too much difference from the one in pipeline **UCDL**. However, the loss, which is computed by the combined loss function, is much higher than the pipelines only using Chamfer distance as the loss function due to Hausdorff distance. Besides, there are some isolated NURBS curves on the front of the duck body, while the distribution of output points is not perfect as pipeline **UCDL**. We assume this phenomenon is caused by Hausdorff distance.

To test the effect of Hausdorff distance, we use an assembly of Hausdorff and Chamfer distance as the loss function in pipeline **UCDH**. As Figure 6.3(g) shows, it depicts duck body perfectly, but duck head part has too much noise. Even without Laplacian loss, the total loss shown in Table 6.1 is very close to the loss in pipeline **UCDHL**, which indicates the Hausdorff distance is large in this case and the Laplacian loss is the real factor in tweaking the pipelines. Therefore, we believe the Hausdorff distance performs poorly under current pipelines.

Besides, in pipeline **UM**, we aim to enforce order in the input point cloud. Figure 6.5 shows the results using pipeline **UM**. The generated duck point cloud is similar to the one from pipeline **UCDL**. However, the horse point cloud, lacking some details, is a bit worse than the one generated in pipeline **UCDL**.

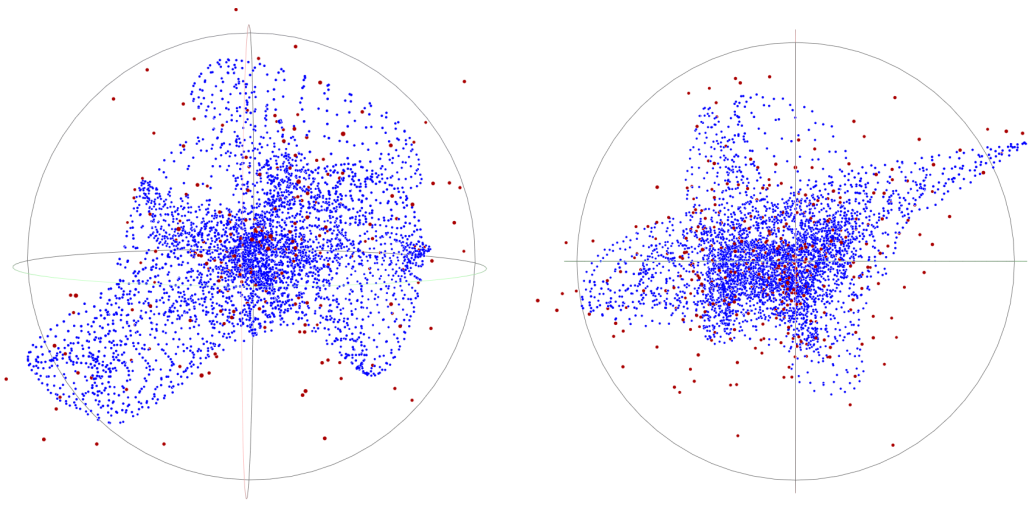
Finally, **UP2P** pipeline shown in Figure 6.6 reconstructs point clouds. As shown in Figure 6.6(a), the reconstructed point cloud of the duck model exhibits a NURBS pattern. It is clearly observed the output follows specific mathematical rules. While the same case can be found in Figure 6.6(b). However, the output is not properly aligned with the input one. This misalignment indicates while the pipeline attempts to fit the data to an underlying NURBS surface, the current implementation still requires further refinement.

We also evaluate mesh reconstruction process using the parameters from the best-fitting output point clouds utilizing geomdl library.



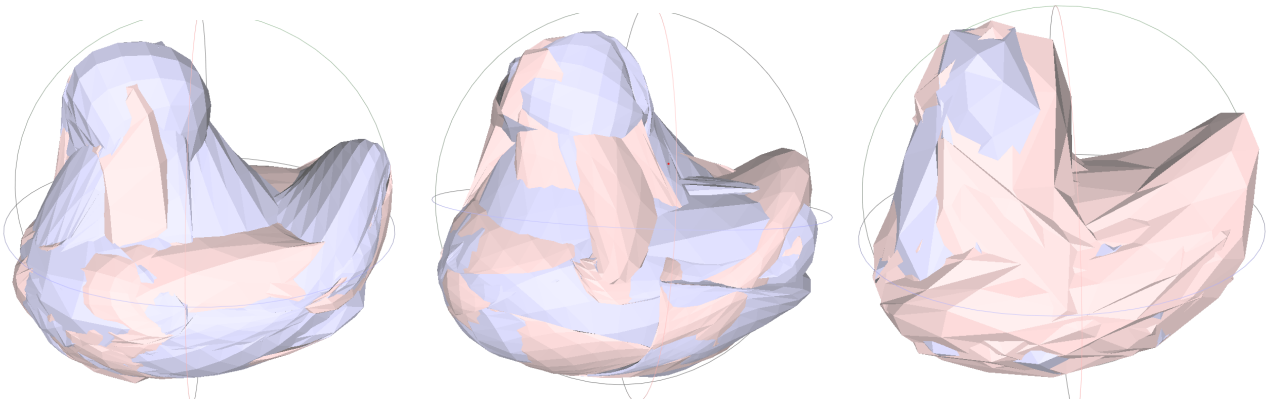
(a) Pipeline UM of the duck model (b) Pipeline UM of the horse model

Figure 6.5: The results of pipeline UM (same configuration as above)



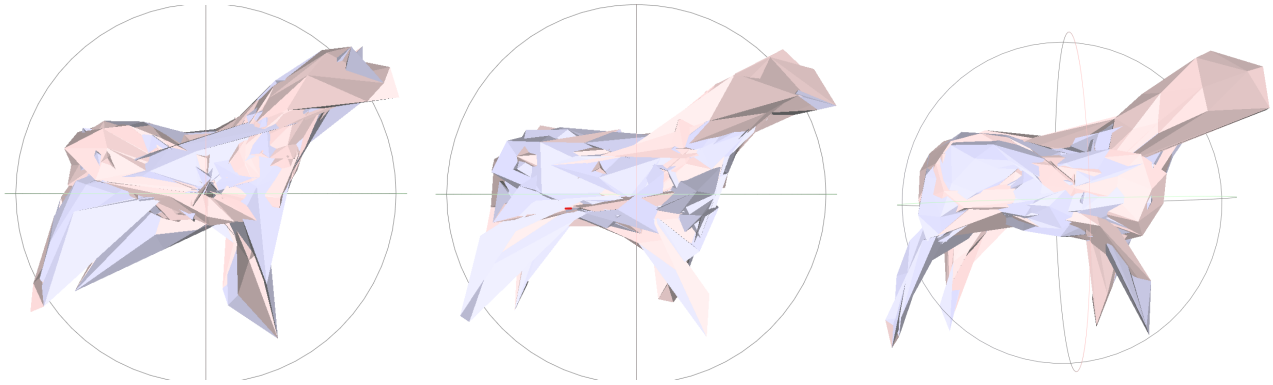
(a) Pipeline UP2P of the duck model (b) Pipeline UP2P of the horse model

Figure 6.6: The results of pipeline UP2P (same configuration as above)



(a) Reconstructed duck model from pipeline UCDL (b) Reconstructed duck model from pipeline UCDHL (c) Reconstructed duck model from pipeline UM

Figure 6.7: Reconstructed duck models from different unsupervised pipelines



(a) Reconstructed horse model from pipeline UCDL (b) Reconstructed horse model from pipeline UCDHL (c) Reconstructed horse model from pipeline UM

Figure 6.8: Reconstructed horse models from different unsupervised pipelines

To evaluate the accuracy of the trained parameters, we reconstruct the duck model using trained parameters from different unsupervised pipelines. As depicted in Figure 6.7, pipeline **UM** produces the duck mesh, though there are several surface overlaps, which is caused by the algorithm used for reconstruction. This confirms the implied point ordering is semi NURBS-based rather than entirely adhering to normal NURBS surfaces. However, in reconstructing the horse model, some surface intersections still occur. Pipeline **UM** still achieves the best performance in this case. Nonetheless, the results are not satisfactory in any case. Although we can get a promising point cloud, the reconstructed meshes are too coarse and irregular. The mesh reconstruction process confirms geomdl combined with our current pipelines may not be a suitable way and we will discuss some alternatives for improvement in the next chapter.

Table 6.3: Quantitative evaluation of duck and horse models

| Pipeline Name | Combined Evaluation Metric | |
|---|---|---|
| | duck model | horse model |
| Pipeline UB | 2.255 | 5.462 |
| Pipeline UC | 1.945 | 4.719 |
| Pipeline UCD | 1.815 | 4.420 |
| Pipeline UCDC | 1.646 | 2.775 |
| Pipeline UCDL | 0.746 | 1.163 |
| Pipeline UCDH | 0.793 | 1.457 |
| Pipeline UCDHL | 1.654 | 2.667 |
| Pipeline UM | 0.7237 | 1.219 |

Additionally, we generate point clouds through parametric sampling from reconstructed meshes and compare them with the original point clouds sampled using Poisson disk sampling. The **Combined Evaluation Metric** $L_{chamfer} + L_{hausdorff} + 0.1 * L_{laplacian}$ is computed to evaluate point cloud and surface quality quantitatively. Table 6.3 shows pipeline **UM** achieves the best evaluation value. Correspondingly, point clouds and meshes reconstructed with the **UM** pipeline are always better than the other approaches as observed. The higher evaluation value a pipeline achieves, the relatively lower quality of point clouds and meshes it can reconstruct. As expected, pipeline **UB**, which computes the L2 distance between point clouds, achieves the highest value and obtain the worst models. Among the other pipelines, **UCDL**, **UCDHL**, and **UM** produce the reconstruction with higher quality.

# 7 Discussion

There are two main sections in this chapter. First, we evaluate the performance of our supervised and unsupervised pipelines by a quantitative and qualitative analysis, respectively. We analyze the best pipelines and worst pipelines on key metrics such as loss and alignment, in order to explore the factors influencing performance. In the following section, we discuss some potential improvements of point cloud and mesh reconstruction based on our current work.

## 7.1 Evaluation

As shown in the previous chapter, we have obtained all results from our supervised pipelines and unsupervised pipelines. In this section, we will synthesize some significant takeaways from the results.

### 7.1.1 Order in the input point cloud

First, ordering property in the input point cloud plays a vital role in the NURBS-Diff module as well as in NURBS mesh reconstruction.

There are 5 different supervised pipelines evaluated: **SB**, **SC**, **SOC**, **SOCH**, and **SOCHL**. As shown in Figure 6.1(b), pipeline **SB**, which computes loss using L2 loss, achieves the lowest loss and best reconstruction. L2 loss is effective here since the supervised pipelines can be considered as a regression problem. Since the input point cloud and the training point cloud follow the same sampling method, the points can be mapped in one-to-one correspondence pairs. While L2 loss is the most efficient way to compute and minimize the difference between them by updating training parameters. To sum up, structured input point clouds guarantee the strong performance of L2 loss, though the other pipeline also works except that they are unable to precisely mimic trained control points as pipeline **SB** among all supervised pipelines.

When it comes to unsupervised pipelines, pipeline **UB** using L2 loss is the worst pipeline as expected, and other pipelines with Chamfer distance, work better. Unlike supervised pipelines, input point clouds in unsupervised cases are typically sampled without any order, while generated point clouds to be trained are always in NURBS-based order. For instance, the wireframe of the input point cloud from the duck model is shown in Figure 3.4. It is clear the point cloud is out of order. Therefore, L2 loss is unsuitable in unsupervised settings as the matching relationship between points is missing. On the contrary, Chamfer distance is less sensitive to point ordering, allowing the pipelines with Chamfer distance to achieve better results.

Furthermore, pipeline **UM** aims to enforce input point cloud in order according to the generated point cloud, since the latter is created through parametric sampling. The result of this pipeline is displayed in Figures 6.5(a) and 6.5(b). The approximation of the duck model is promising. Although it lacks some finer details regarding the horse model, generally it performs well by partially retrieving the ordering in the input point cloud. This suggests when the input conforms reasonably to a parametric structure, it is feasible to perform a successful point cloud reconstruction with the NURBS-based pattern.

### 7.1.2 Laplacian loss in unsupervised pipelines

Second, Laplacian loss, which refines the positions of predicted control points, is extremely useful in unsupervised pipelines.

As displayed in Figure 6.3(c), Chamfer distance alone (**UC**) is not enough to achieve a qualified approximation of ground truth since there exists too much noise, and trained control points are roaming away from the trained point cloud. Chamfer distance is just to align two point clouds on best efforts. Therefore, in pipeline **UCDC**,

we set the initial control points to be cylinder-based, which is to cover ground-truth points inside and constrain training to that region. However, although it achieves the minimum loss and performs much better than pipeline **UC**, the training process remains uncontrollable since the use of Chamfer distance and the unordered input point cloud. By evaluating the result shown in Figure 6.3(e), although noise is reduced, control points do not show any pattern.

So in pipeline **UCDL**, we combine Chamfer and Laplacian loss since reducing Laplacian loss refines predicted control points, evidenced by Figure 6.3(f). As shown in the figure, the predicted control points are tight to output point cloud and more and more close to ground-truth control points. Most importantly, the trained point cloud also reveals the NURBS surface pattern. In this case, although the total loss of this pipeline is higher than pipeline **UCDC**, the **Combined Evaluation Metric** is almost the best among unsupervised pipelines.

### 7.1.3 Mesh reconstruction with retrieved parameters from unsupervised pipelines

Moving forward, we illustrate the reconstruction of the duck and horse models using trained parameters from our best unsupervised method. With the help of geomdl, we can generate NURBS surfaces as shown in Figure 6.7 and 6.8. The generated models lack finer details since the coarse triangulation results from the unordered or partially ordered output point clouds.

Even though the output point cloud represents the sampling from a NURBS surface, however, the ordering of points is only approximate rather than fully matching the ground truth using parametric sampling, as seen in supervised training. Consequently, the generated NURBS surface is not as smooth as the original model. This suggests an alternative way to reconstruct meshes may be needed.

In conclusion, the most effective unsupervised pipelines are **UM** and **UCDL** with Chamfer and Laplacian loss. The Chamfer distance is applied to align two point clouds and Laplacian loss is used to keep control points stick to each other and enable control points or output point clouds in the partial order, while pipeline **UM** aims to retrieve the order in input point clouds. Although both pipelines achieve higher losses than pipeline **UCD**, they are the most stable and clean ones, which correspond to lower combined evaluation metric shown in Table 6.3.

## 7.2 Future work

### 7.2.1 Sampling method adjustment

In current unsupervised pipelines, we sample point clouds directly from mesh using Poisson disk sampling. However, this approach produces an unordered set of points that lacks structured connectivity. Poisson disk sampling fails to preserve the topological information contained in NURBS surfaces. By comparing with supervised pipelines, the importance of parametric sampling has been established.

Therefore, we need to find a better sampling method to order points in point cloud, which aims to bridge the gap between unordered point sets and the structured representations necessary for NURBS-based shape recovery. One approach is to conduct permutation to the control point set or input point cloud to get a parametric sampling as shown in Figure 6.1(a).

Upon examining the reconstruction results in Figure 6.3, it can be observed some details in the reconstruction process are missing. For instance, the ears of the horse model are not fully reconstructed after training. Through further analysis, it should result from insufficient sampling density in certain regions of the input point cloud. Areas with complex geometry, like the ears in the horse model, should benefit from a denser distribution of sampled points to adequately capture the shape details during optimization.

### 7.2.2 Mesh subdivision or point cloud segmentation

As the figures shown in Appendix A, we acknowledge variant pipeline **UCDL** works better when the shapes are simple. We can perform pro-processing methods like mesh subdivision or point cloud segmentation to split data into simple mesh or point cloud and run the corresponding pipeline. We also notice other pipelines using NURBS-Diff module also works better when dealing with simple objects, like cylinders or sheets. However, there

may be an existing issue that the boundary of each part may not be smooth enough. Thus we need to solve the existing problem as well.

### 7.2.3 Reconstruction with mesh

As observed in Figures 6.7 and 6.8, the reconstruction of NURBS surfaces using geomdl is not perfect since the algorithm assumes the surface can be fully defined by trained parameters, however, this is not the case. The rendering method in geomdl is too simple to use, while in unsupervised settings there should exist some noise and unordered points which bias from normal NURBS surface pattern. So we may need to utilize other methods rather than geomdl to reconstruct meshes from unstructured point clouds.

# 8 Conclusion

In this thesis, we explore the solutions to reconstruct NURBS surface models using supervised and unsupervised settings. We propose several different pipelines mainly trained through the NURBS-Diff module to tackle this challenging task.

We begin by providing technical background. The key concepts, such as point cloud representation, Poisson disk sampling, and NURBS surface are introduced in this chapter. A literature review of related work is also covered. We mainly discuss the supervised pipelines in ParSeNet [SLK+20] and [DBS+22].

Afterwards, We clarify the obstacles when working with unsupervised settings. The main issue lies in unstructured input point clouds while the original NURBS-Diff module is sensitive to ordering property. Afterwards, we provide a detailed explanation of the NURBS-Diff module and discuss the point-to-surface approximation algorithm.

In the next chapter, we experiment extensively with a variety of loss functions, sampling mechanisms, and network architectures using supervised and unsupervised pipelines. The supervised pipelines leverage ordering information between inputs and outputs. While unsupervised pipelines, our main focus, aims to find a feasible solution to approximate target point cloud or surface without such correspondences.

Through different pipelines, we obtain results evaluating the effectiveness of corresponding pipelines. The results are two main parts, reconstructed point clouds and surfaces, respectively. We analyze supervised results first. Next, we propose a metric for qualitative evaluation regarding unsupervised pipelines, from which pipeline **UM** and **UCDL** outperform in comparison to other pipelines which correspond to the result from the quantitative evaluation.

Furthermore, we analyze the differences between supervised pipelines and unsupervised pipelines. We also analyze the reason why pipeline **UM** and **UCDL** work better,

In summary, this study can be considered as an extension of the NURBS-Diff module by introducing novel pipelines. Unstructured point clouds, sampled either from a NURBS surface or other meshes, are feasible to retrieve NURBS-based point clouds via our pipelines. The findings advance the step towards the reconstruction of NURBS-based meshes from raw point clouds.

# Acknowledgements

The research was conducted at the VMM lab in the IFI department of UZH. I would like to express my sincere gratitude to Prof. Dr. Renato Pajarola for giving me this precious opportunity to participate in this interesting project. I gratefully appreciate his constructive suggestions for the design of network architecture.

I also want to express my sincere gratitude to Lizeth J. Fuentes Perez, for her invaluable guidance and suggestions throughout this research. I am deeply appreciative of her willingness to share her technical knowledge and general experience.

I would also like to thank my family, especially my parents, for their endless love and support. Finally, I am thankful to my friends, who provided many valuable suggestions whenever I encountered difficulties in my work.

# Bibliography


[BK19] Onur Rauf Bingol and Adarsh Krishnamurthy. NURBS-Python: An open-source object-oriented NURBS modeling framework in Python. *SoftwareX*, 9:85–94, 2019.

[BM22] Alexandre Boulch and Renaud Marlet. Poco: Point convolution for surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6302–6314, June 2022.

[Bri07] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches*, 10(1):1, 2007.

[CC78] Edwin E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Seminal graphics: pioneering efforts that shaped the field*, 1978.

[CCC+08] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[DBS+22] Anjana Deva Prasad, Aditya Balu, Harshil Shah, Soumik Sarkar, Chinmay Hegde, and Adarsh Krishnamurthy. Nurbs-diff: A differentiable programming module for nurbs. *Computer-Aided Design*, 146:103199, 2022.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[Jon06] Thouis R. Jones. Efficient generation of poisson-disk sampling patterns. *Journal of Graphics Tools*, 11(2):27–36, 2006.

[Ma05] Weiyin Ma. Subdivision surfaces for cad—an overview. *Computer-Aided Design*, 37(7):693–709, 2005.

[PGK02] M. Pauly, M. Gross, and L.P. Kobbelt. Efficient simplification of point-sampled surfaces. In *IEEE Visualization, 2002. VIS 2002.*, pages 163–170, 2002.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PT97] Les Piegl and Wayne Tiller. *The NURBS Book (2nd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1997.

[SLK+20] Gopal Sharma, Difan Liu, Evangelos Kalogerakis, Subhransu Maji, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds, 2020.

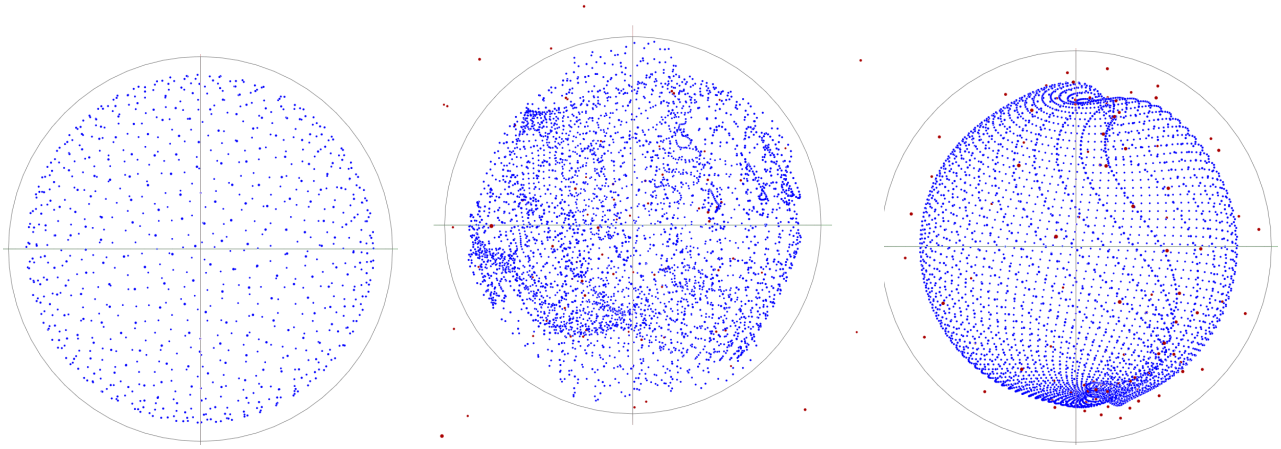
[SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit (4th ed.)*. Kitware, 2006.

[SRN08] M.M.M. SARCAR, K.M. RAO, and K.L. NARAYAN. *Computer Aided Design and Manufacturing*. PHI Learning, 2008.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

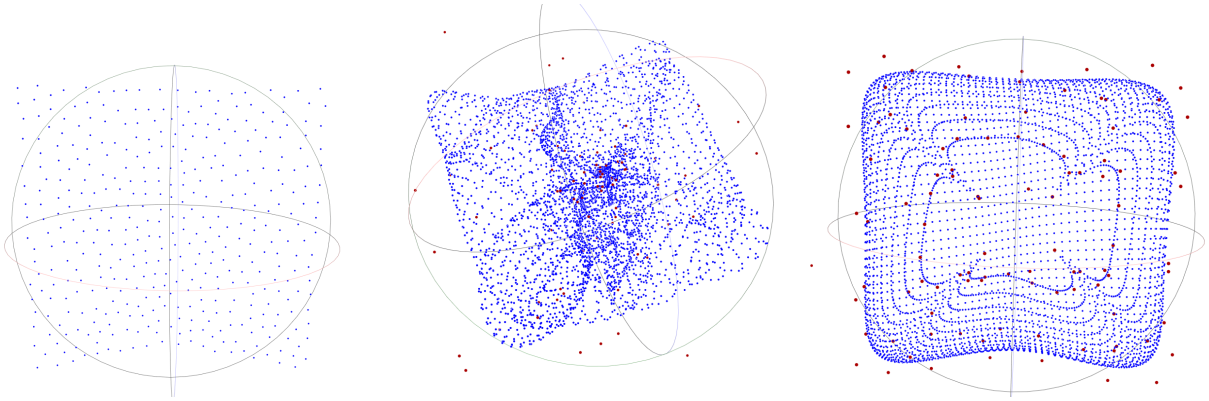
[WW91] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. Association for Computing Machinery, New York, NY, USA, 1991.

# Appendix A: the rest of results



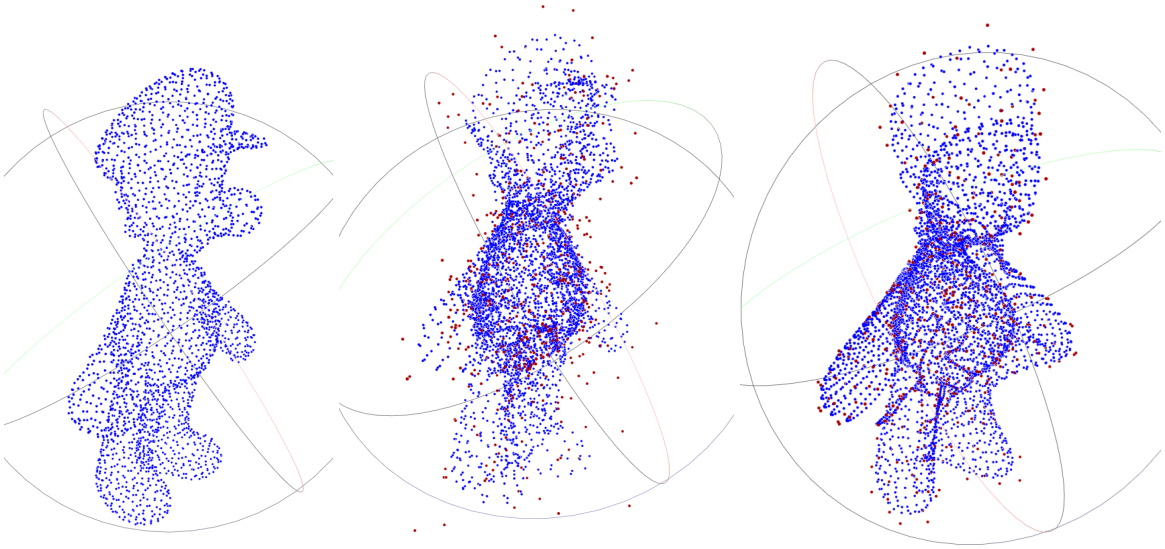
(a) Poisson sampling of the sphere model (b) Normal Pipeline UCDL of the sphere model (c) Variant Pipeline UCDL of the sphere model

Figure 1: The point cloud of the sphere model



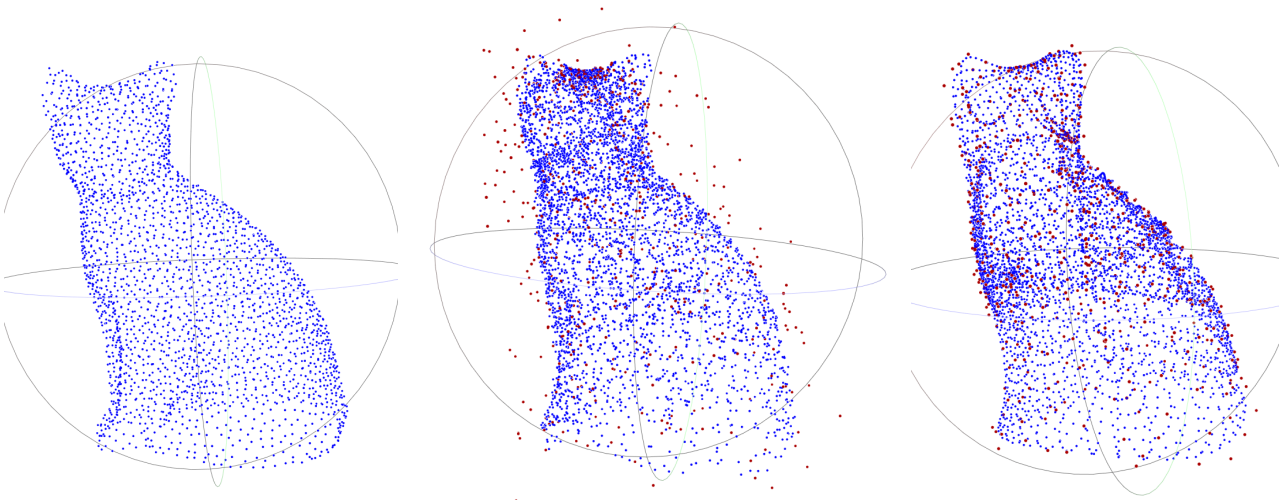
(a) Poisson sampling of the plane model (b) Normal pipeline UCDL of the plane model (c) Variant pipeline UCDL of the plane model

Figure 2: The point cloud of the plane model



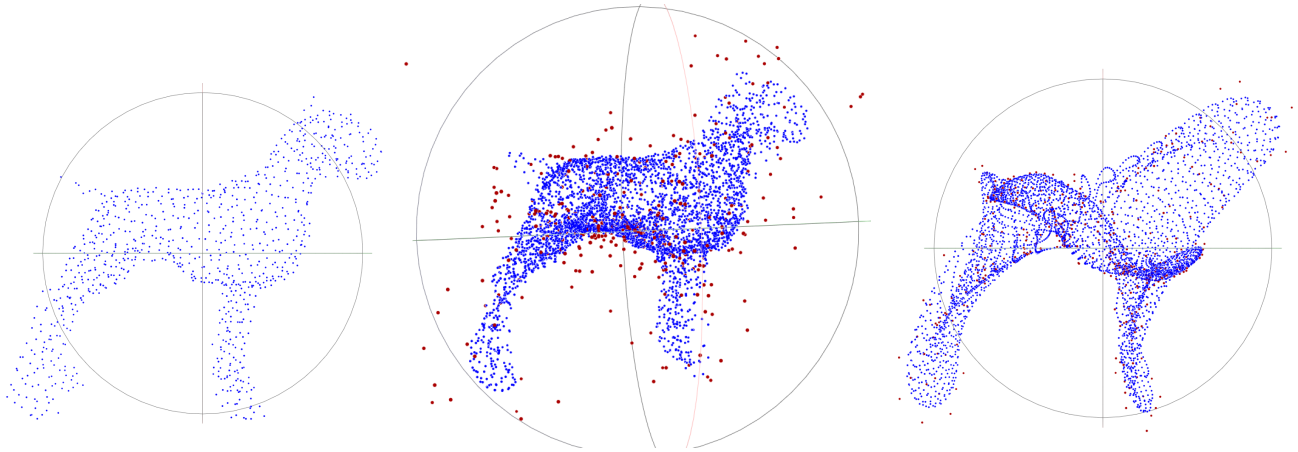
(a) Poisson sampling of the luigi model (b) Normal pipeline UCDL of the luigi model (c) Variant pipeline UCDL of the luigi model

Figure 3: The point cloud of the luigi model



(a) Poisson sampling of the cat model (b) Normal Pipeline UCDL of the cat model (c) Variant Pipeline UCDL of the cat model

Figure 4: The point cloud of the cat model



(a) Poisson sampling of the hound model (b) Normal pipeline UCDL of the hound model (c) Variant pipeline UCDL of the hound model

Figure 5: The point cloud of the hound model

As discussed in Chapter 5, normally, Laplacian loss in pipeline **UCDL** is applied to the output point. We refer to this pipeline as the "normal pipeline **UCDL**". However, Laplacian loss can be applied to the predicted control points as well. When the Laplacian is applied to the predicted point cloud, we refer to this variant approach as "variant pipeline **UCDL**".

We display Figures 1 to 5, comparing the performance of the normal and variant pipeline **UCDL**. In each figure, subfigure on the right shows the input point cloud by Poisson disk sampling. The middle one is the reconstructed point cloud by normal pipeline **UCDL**. While the right subfigure displays the result of variant pipeline **UCDL**. As we can observe from these figures, the variant pipeline reconstructs point clouds with a more explicit NURBS surface pattern. This pipeline outperforms the normal one, especially on simple geometries, such as the sphere and the plane shown in the figures. Furthermore, the most important finding is the predicted control points are tightly close to the output point cloud. However, the approach performs average on complex objects since it does not fully learn high-level details from the surface, which may indicate the number of predicted control points is insufficient for complex geometry reconstruction. To address this, we have several approaches to tweak the pipeline.

Subdividing the original mesh as subdivision objects do or Segmenting the input point cloud should be helpful to mitigate this issue. Besides, increasing the number of control points while keeping degrees and weights the same should gain more control power on complex shapes. Moreover, combining the losses of these two pipelines may also be feasible for the reconstruction. These potential tweaks, considered as our future work, serve as a stepping stone for further investigation into how to unleash the capabilities of pipeline **UCDL**.