Technische Universität München

Computational Science and Engineering
(Int. Master's Program)

Master Thesis

Investigation and Implementation of Non-Matching Grids Data Transfer

Tianyang Wang

1st examiner:	Univ.-Prof. Dr.-Ing. Kai-Uwe Bletzinger
2nd examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Supervisor:	Majid Hojjat, M.Sc.
Organization:	Lehrstuhl für Statik (Prof. Bletzinger)
Thesis handed in on:	December 13, 2010

Abstract

Partitioned solution of multiphysics problems often involves the data transfer between non-matching grids. This work studies different mapping methods for transferring data between non-matching grids in surface-coupled problems. The simplest method is "nearest neighbor mapping", in which the data value on the query node is assigned by the data value on its nearest neighbor. "Barycentric interpolation mapping" and "NURBS surface interpolation mapping" make an improvement by interpolating the data values on more than one neighboring nodes. Another two mapping methods studied in this thesis are "radial basis function interpolation mapping" and "Kriging mapping", both use the data values on all nodes to build a function model describing the data field within a domain. These mapping methods are tested under different test cases, the mapping error, the computation time and the smoothness of the data field are presented and compared.

Acknowledgment

Here I want to express my gratitude to the people who offered me help during this work.

I thank Majid Hojjat to be the supervisor of my thesis. Without his planning and instructions during the work, this thesis would not reach a good end. And his proofreading of my thesis is a large support.

I am thankful to Dr. Roland Wüchner for offering me the topic and taking care of the organization. And I want to thank the people from the chair of Statik for the supporting environment.

Finally, I thank Prof. Kai-Uwe Bletzinger and Prof. Hans-Joachim Bungartz for their efforts to examine this work.

Contents

1 Introduction

Multiphysics is a very hot topic in computational science and engineering community. One of the common solving strategies uses a partitioned way, which means different physical fields are solved separately. One important step in such strategy is the data exchange (e.g. displacements, pressures, ..., in the case of fluid-structure interaction) across the interface between different physical fields. However, different physical fields might have their own discretization/meshing methods on the interface, the nodes/grids from different fields are not coincided which is shown in Figure 1.1. As a result, we cannot directly assign the data value from one side to the other side. Therefore, the data transfer between non-matching grids requires special treatment, which is studied in this thesis.

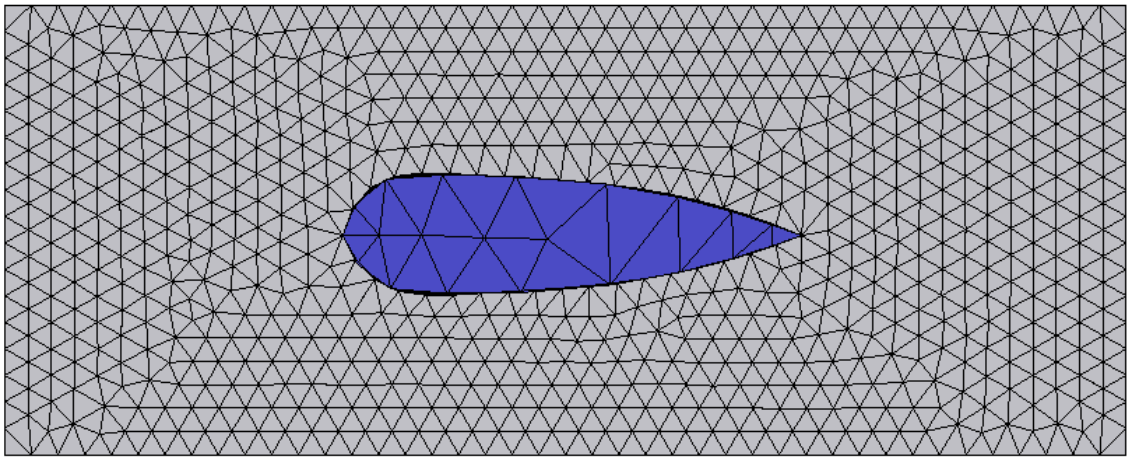


Figure 1.1: The modeling of the fluid-structure interaction between the airfoil and the air around it. The blue area is the structure model of the airfoil and the gray area represents the air. At the boundary of the airfoil, the nodes from both side are not coincided because of the different element sizes.

1.1 Data Transfer Model

Because the data transfer only happens at the interface between different fields, it is not necessary to take the whole body of the field into account. The problem is simplified by the data transfer across one surface (in this thesis, it is assumed that the physical fields exist in 3D space so the interface between them is a surface). The data transfer model is that, two different meshes are obtained by meshing the same surface in different

refinement, and the task is to transfer the discretized data from one mesh to the other mesh. This can be expressed more in a mathematical way as, given a set of points in a region with given data values on them, we want to compute the data values on another set of points in this region. This process is also referred as *mapping*, often with interpolation and projection techniques involved. In the following, the word *mapping* is used more often as a substitute of *data transfer*.

1.2 Chapter Overview

In Chapter 2, the theory of five different mapping methods will be introduced. The main work of this thesis is the implementation of the C++ program *MeshMapping* which performs mapping with the methods introduced in Chapter 2. And Chapter 3 explains the code of the program *MeshMapping* and how it works. In Chapter 4, the program *MeshMapping* is run under a big number of test cases, in order to analyze the performance of each mapping method. In Chapter 5, the work of this thesis is concluded with the future work for improvement.

2 Mapping Methods

Mapping the data from one mesh to the other mesh, is the process of computing the data values on a set of points in a region by another set of points with given data values in this region. In the following, we assume that the data are mapped from mesh **A** to mesh **B**, i.e. the data values on points in **A** are given and the data values on points in **B** are going to be computed by mapping.

In this chapter, five different mapping methods will be introduced. The common characteristic of all mapping methods is that, the data on **B** are computed one point by one point. The point on which the data value is to be computed is called the *query point*. The essence of all mapping methods is how to compute the data value on the query point based on the information of the points with given data values. This is illustrated by Figure 2.1.

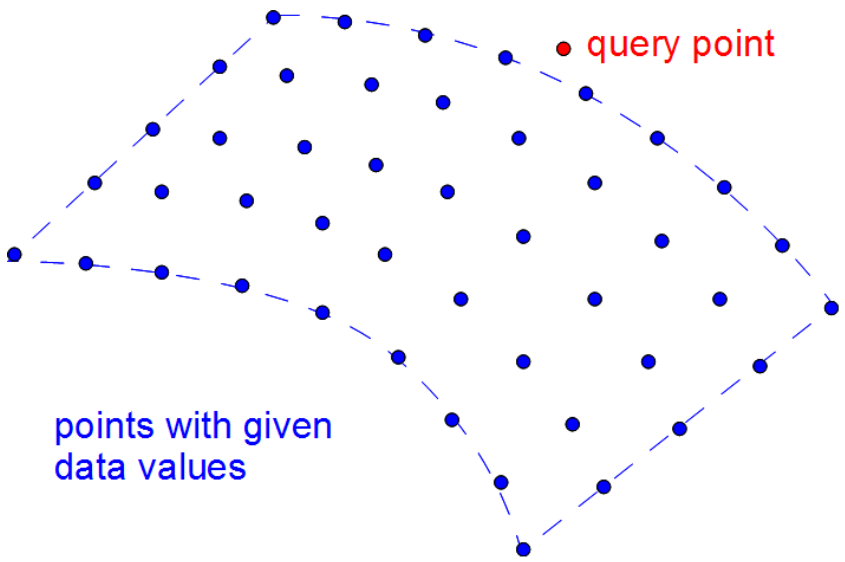


Figure 2.1: The data value on the query point is going to be computed based on the information of the data values on a set of points.

The simplest mapping method is *nearest neighbor mapping*, where the data value on the query point is assigned to be the same as the data value on its nearest neighboring point. The accuracy of this method is limited because only the information on one point is used. *Barycentric interpolation mapping* and *NURBS surface interpolation mapping* make an improvement by constructing a geometry by a number of neighboring points. We have to find the projection of the query point on this geometry and then interpolate the data value on the projection point. The data value on the projection point is regarded

as a better guess of the data value on the query point than that on the nearest neighbor. In both *radial basis function (RBF) interpolation mapping* and *Kriging mapping*, an approximate function of the data field on the surface ($f(x, y, z)$) is constructed. Then by inputting the position of the query point to the function, the data value can be directly computed.

At the end of the introduction to each mapping method, the complexity of the mapping algorithm will be presented. It is dependent on the number of points in both **A** and **B**. For convenience, N_A is used to represent the number of points in **A** and N_B is used to represent the number of points in **B**.

2.1 Nearest Neighbor Mapping

Nearest neighbor mapping is the simplest mapping method. The query point searches for the nearest neighboring point, and the data value on the query point is assigned to be the same as the data value on the nearest neighbor. Figure 2.2 is an illustration of nearest neighbor mapping.

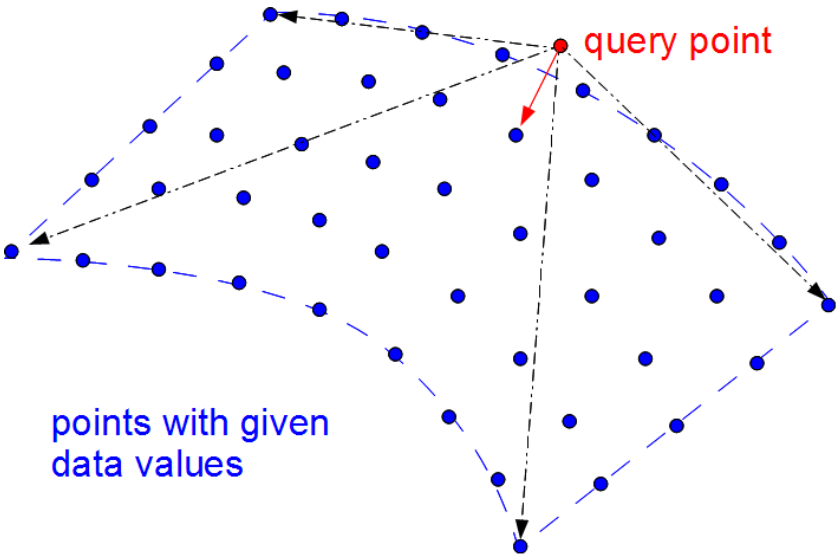


Figure 2.2: Nearest neighbor mapping. The red arrow points to the nearest neighbor.

2.1.1 Nearest Neighbor Searching

The most important part in nearest neighbor mapping is *nearest neighbor searching*. A straightforward algorithm is the *brute force method*. In such method, all points in **A** are visited to compute the distances to the query point, then the point with the minimum distance is defined as the nearest neighbor. Since all points in **A** are visited once, so the complexity of the brute force method is $O(N_A)$. And because all points in **B** do nearest neighbor searching once, so the total complexity of nearest neighbor mapping with brute force method is $O(N_B \cdot N_A)$.

A more efficient algorithm is the searching with *kd-tree* (short for k-dimensional tree) data structure. The kd-tree data structure is based on a recursive subdivision of space into disjoint hyper-rectangular boxes [2]. This is shown in Figure 2.3. Each node in the tree corresponds to a box which contains a number of points. The root node in the tree corresponds to the bounding box of all points. The tree is built by a recursive algorithm as, if there are more than one points in a box, then the box will be split into two sub-boxes by a hyperplane, so that two child nodes are added to the tree.

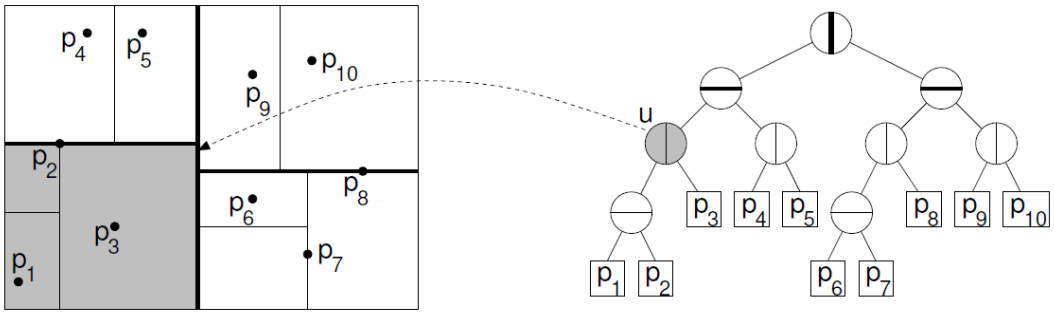


Figure 2.3: A kd-tree of bucket-size one and the corresponding spatial decomposition [2].

The algorithm of nearest neighbor search in a kd-tree operates recursively. When a node (except for the leaf node) is visited, the first step is to visit the child node that is closer to the query point. The "closer" is determined by to which side of the hyperplane the query point belongs. Then this child node is visited in a recursive way. If the child node is a leaf node, then the point (only one) corresponding to this node is regarded as the nearest point. On return, it is to be decided whether to visit the other child node or not. If it is impossible to have a smaller distance in the other child node than the nearest distance so far (by checking the dimension of the box of the other child node), then the other child node is skipped, otherwise, it is visited recursively.

2.1.2 ANN Library

In this work, the nearest neighbor searching is realized by using the library ANN [1] which is an open source library dealing with the nearest neighbor problem. ANN stands for the *Approximate Nearest Neighbor* library. Although the library provides several special options to make improvement in the running time, only the kd-tree with a standard search is used. The constructor and the search function of a kd-tree in ANN library is listed below:

```
ANNkd_tree::ANNkd_tree(
        ANNpointArray pa,          // data point array
        int n,                     // number of points
        int d);                    // dimension
```

```
void ANNkd_tree::annkSearch(
        ANNpoint q,                 // query point
        int k,                      // number of near neighbors to find
        ANNidxArray nn_idx,         // nearest neighbor array
        ANNdistArray dists,         // distance to near neighbors
        double eps=0.0);            // error bound
```

2.1.3 Complexity

When building the kd-tree, the entire set of the points must be visited at each level of the kd-tree, and because the depth of the kd-tree is $O(\log N_A)$, the total complexity of building a kd-tree is $O(N_A \cdot \log N_A)$ [3]. The complexity of searching in a kd-tree is $O(\log N_A)$ [3]. Therefore, the total complexity of nearest neighbor mapping using kd-tree is $O(N_A \cdot \log N_A) + O(N_B \cdot \log N_A)$, or simply $O(N \cdot \log N)$ when $N_A \approx N_B = N$. It is better than using brute force method which has a complexity of $O(N_B \cdot N_A)$ or $O(N^2)$ when $N_A \approx N_B = N$.

2.2 Barycentric Interpolation Mapping

In barycentric interpolation mapping, the nearest *three* points of the query point are searched. These three points construct a triangle in space. The query point is projected to the triangle plane to get its projection point, and then barycentric interpolation is performed on the projection point to compute the data value on it. The data value of the projection point is the guess of the data value on the query point. This is shown in Figure 2.4.

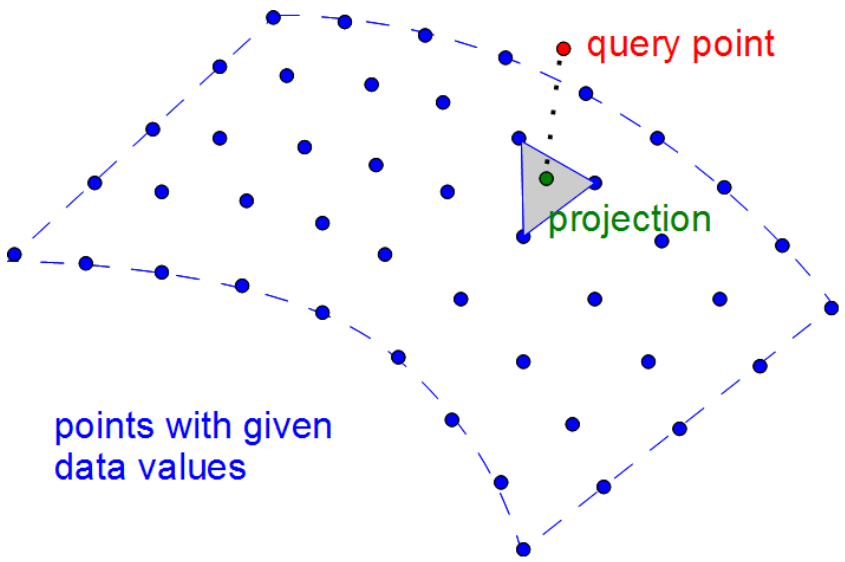


Figure 2.4: Barycentric interpolation mapping.

In barycentric interpolation mapping, the kd-tree data structure introduced in Section 2.1 is used to perform the nearest neighbor search of the three nearest points. So the problem left is how to do projection and barycentric interpolation.

2.2.1 Projection Formula

Assume there are three points constructing a triangle, which are P_1, P_2 and P_3. The coordinates of these points are (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) respectively. There is another point P_0 which has coordinates (x_0, y_0, z_0), is going to be projected onto the plane constructed by P_1, P_2 and P_3. The projection point is denoted by P_4 which has coordinates (x_4, y_4, z_4). From basis linear algebra, it is easy to deduct the formula that is used to calculate the projection of a point onto a triangle. The formula turns out to be a linear equation system, the unknown vector is $(x_4, y_4, z_4)^T$, the left hand side (LHS) is

$$\begin{pmatrix} x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \\ \begin{vmatrix} y_2 - y_1 & z_2 - z_1 \\ y_3 - y_1 & z_3 - z_1 \end{vmatrix} & - \begin{vmatrix} x_2 - x_1 & z_2 - z_1 \\ x_3 - x_1 & z_3 - z_1 \end{vmatrix} & \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix} \end{pmatrix},$$

and the right hand side (RHS) is

$$\begin{pmatrix} (x_2 - x_1)x_0 & + & (y_2 - y_1)y_0 & + & (z_2 - z_1)z_0 \\ (x_3 - x_1)x_0 & + & (y_3 - y_1)y_0 & + & (z_3 - z_1)z_0 \\ \begin{vmatrix} y_2 - y_1 & z_2 - z_1 \\ y_3 - y_1 & z_3 - z_1 \end{vmatrix} x1 & - & \begin{vmatrix} x_2 - x_1 & z_2 - z_1 \\ x_3 - x_1 & z_3 - z_1 \end{vmatrix} y1 & + & \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix} z1 \end{pmatrix}.$$

So the global equation system is

$$LHS \cdot (x_4, y_4, z_4)^T = RHS. \tag{2.1}$$

It can be solved by for example by Gauss elimination so that the projection point can be obtained.

2.2.2 Barycentric Interpolation

The process of barycentric interpolation is as follows: Given a triangle with end points P_1, P_2 and P_3, and a point P_4 on the triangle plane, barycentric interpolation provides a way to interpolate the data value on P_4 by interpolating the data values on P_1, P_2 and P_3. Assume v_i represents the data value on point P_i, with $(i = 1, 2, 3, 4)$, by barycentric interpolation we have

$$v_4 = w_1 v_1 + w_2 v_2 + w_3 v_3, \tag{2.2}$$

with

$$w_1 = (\pm)\frac{area\ of\ \triangle P_4P_2P_3}{area\ of\ \triangle P_1P_2P_3},$$
$$w_2 = (\pm)\frac{area\ of\ \triangle P_4P_1P_3}{area\ of\ \triangle P_1P_2P_3},$$
$$w_3 = (\pm)\frac{area\ of\ \triangle P_4P_1P_2}{area\ of\ \triangle P_1P_2P_3}. \tag{2.3}$$

Whether the weights $w_i (i = 1, 2, 3)$ are positive or negative, depends on where P_4 locates. If P_4 and $P_j (j = 1$ or 2 or 3) locate at the same side of the other two end points, use the positive sign for w_j, otherwise, use the negative sign (see Figure 2.5). This ensures that the sum of the weights is always equal to one.

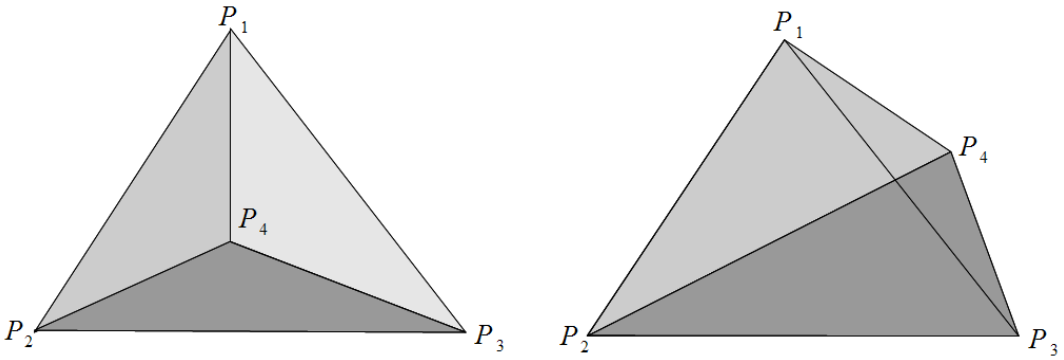


Figure 2.5: Barycentric interpolation. In the left case, since P_4 locates inside the triangle, all the weights are positive. In the right case, P_4 and P_2 locate at the opposite side of the line P_1P_3, so w_2 is negative while the other two weights are positive. However in both cases, the sum of the weights is one.

2.2.3 Complexity

When computing the data value, the first step is to search three nearest points in $\mathbf{A}$, which is of complexity $O(\log N_A)$ as shown in Section 2.1. Then projection and barycentric interpolation will be performed in the second step. It is easy to see the complexity of the this step is independent of N_A, so it can be ignored. Then the total complexity of barycentric interpolation mapping is the same as nearest neighbor mapping which is $O(N_A \cdot \log N_A) + O(N_B \cdot \log N_A)$ or $O(N \cdot \log N)$ when $N_A \approx N_B = N$.

2.3 NURBS Surface Interpolation Mapping

NURBS is an abbreviation of *non-uniform rational B-spline*, which is a mathematical model that can be used to construct curves and surfaces. In NURBS surface interpolation mapping, we first search for the nearest neighbor, and then construct a NURBS surface

which interpolates the nearest neighbor and a number of points around it (which is called *NURBS Surface Interpolation*). Then we project the query point on the NURBS surface to get its projection point. The property of NURBS surface that enables it to be applied in mapping is that, the same law of interpolating NURBS surface by geometrical points can be applied to the data values on the points. This property is widely applied in isogeometric analysis. Then the data value on the projection point can be obtained, which is the guess of the function value on the query point. The process is shown in Figure 2.6.

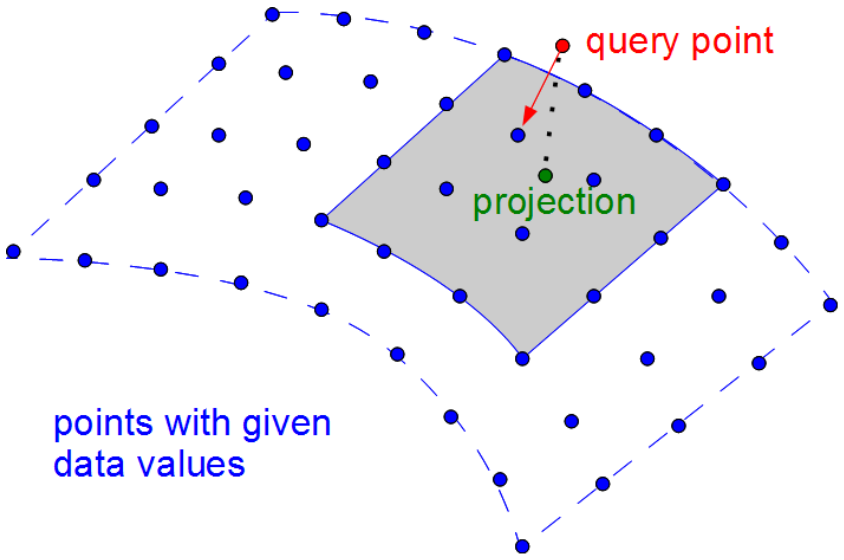


Figure 2.6: NURBS surface interpolation mapping. The red arrow points to the nearest neighbor and the dark surface is the NURBS surface patch with 4×4 points.

The drawback of this mapping method is that, the NURBS surface can be built only by griding points, so in the implementation of this mapping method it is required that **A** is of element type *four nodes' quadrilateral*.

In this section, the basic knowledge of NURBS is introduced and the interpolation and projection with NURBS surface is presented. The knowledge presented in this section is learned from [6]. One remark is that the interpolation is actually performed by *non-rational* B-spline surface instead of NURBS (*rational B-spline*) surface. However, B-spline surface is a special type of NURBS surface, so "NURBS surface" is used in this thesis.

2.3.1 Defining the Patch

Defining the patch means defining which group of points that the NURBS surface will interpolate. The patch is defined by the nearest neighbor of the query point and the points surrounding the nearest neighbor. There is special algorithm on selecting the surrounding points, which will be introduced later in Chapter 3.

2.3.2 B-spline Curves and Surfaces

A nondecreasing sequence of real numbers $U = u_0, \ldots, u_m$ is called the *knot vector*, with u_i is called the *knot*. Then the B-spline basis function of pth-degree $N_{i,p}(u)$ is defined as

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$
$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u). \tag{2.4}$$

It is easy to see that $N_{i,p}(u)$ is defined on the knot span $[u_i, u_{i+p+1}]$, so each B-spline basis function of p-degree corresponds to $p + 2$ knots.

Given $n + 1$ points in space, a pth-degree B-spline curve can be formulated by the linear combination of these points and the B-spline basis functions, which is

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i \qquad 0 \leq u \leq 1, \tag{2.5}$$

where $\mathbf{P}_i$ is called the *control point*. Usually it is a vector containing the coordinates of a point. But for applying NURBS into mapping, we make it also contain the data values on the point, we can use eq.(2.5) not only to compute a point on the curve, but also to compute the data values on the point. Note that the B-spline curve does not pass all the control points, so it is not an interpolation technique. We will introduce the interpolation with NURBS curve and NURBS surface later.

To ensure that the number of control points are equal to the number of p-degree B-spline basis functions, it must be satisfied that

$$m = n + p + 1. \tag{2.6}$$

We can compute the knot vector in a *equally spaced* way: If the number of the control points is $n + 1$, and a p-degree B-spline curve is desired, then the first $p + 1$ knots are set to 0 while the last $p + 1$ knots are set to 1, and knots in between are set to $\frac{1}{n+1-p}, \frac{2}{n+1-p}, \ldots, \frac{n+p}{n+1-p}$ sequentially. An example of computing the knot vector with the same six control points is as below:

$$\begin{aligned} p = 1 \quad &: \quad U = \{0, 0, 1/5, 2/5, 3/5, 4/5, 1, 1\} \\ p = 2 \quad &: \quad U = \{0, 0, 0, 1/4, 2/4, 3/4, 1, 1, 1\} \\ p = 3 \quad &: \quad U = \{0, 0, 0, 0, 1/3, 2/3, 1, 1, 1, 1\} \\ p = 4 \quad &: \quad U = \{0, 0, 0, 0, 0, 1/2, 1, 1, 1, 1, 1\} \\ p = 5 \quad &: \quad U = \{0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1\} \end{aligned}$$

After the knot vector is computed, the B-spline basis functions can be computed by eq.(2.4), and then the B-spline curve can be obtained by eq.(2.5).

From eq.(2.5) we can observe that only $N_{i,p}(u)$ is dependent of u, so the kth derivative of $\mathbf{C}(u)$ which is denoted by $\mathbf{C}^{(k)}(u)$ can be derived as

$$\mathbf{C}^{(k)}(u) = \sum_{i=0}^{n} N_{i,p}^{(k)}(u)\mathbf{P}_i. \tag{2.7}$$

And because $N_{i,p}(u)$ is a simple polynomial, the kth derivative of it can be formulated easily, which is omitted here.

A B-spline surface is obtained by taking a bidirectional net of control points, two knot vectors, and the products of the univariate B-spline functions as [6]

$$\mathbf{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)\mathbf{P}_{i,j}. \tag{2.8}$$

The knot vectors and the B-spline functions can be obtained just as the univariate case.

And the derivatives of the B-spline surface can be obtained by the derivatives of the basis functions as

$$\frac{\partial^{k+l}}{\partial^k u \partial^l v}\mathbf{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}^{(k)}N_{j,q}^{(l)}\mathbf{P}_{i,j}. \tag{2.9}$$

The derivatives will be used later in finding the projection point.

2.3.3 B-Spline Surface Interpolation

First, let us look at the curve interpolation using NURBS, then it is easy to generalize to the surface interpolation.

Given a set of points $\mathbf{Q}_k$, $k = 0, \ldots, n$, if there is a pth-degree B-spline curve interpolates these points, then we have $n+1$ equations

$$\mathbf{Q}_k = \mathbf{C}(\bar{u}_k) = \sum_{i=0}^{n} N_{i,p}(\bar{u}_k)\mathbf{P}_i, \tag{2.10}$$

which are obtained by substituting $u = \bar{u}_k$ in eq.(2.5). Up to now, both $\bar{u}_k$ and $\mathbf{P}_i$ are unknown. In fact $\bar{u}_k$ can be computed heuristically by

$$\bar{u}_0 = 0 \qquad \bar{u}_n = 1$$
$$\bar{u}_k = \bar{u}_{k-1} + \frac{\|\mathbf{Q}_k - \mathbf{Q}_{k-1}\|}{d} \qquad k = 1, \ldots, n-1, \tag{2.11}$$

with

$$d = \sum_{k=1}^{n} \|\mathbf{Q}_k - \mathbf{Q}_{k-1}\|, \tag{2.12}$$

where $\|.\|$ represents the Euclidean distance. Because the *equally spaced* knot vector can result in singular system [6], the knot vector is computed in another way as

$$u_0 = \ldots = u_p = 0 \qquad u_{m-p} = \ldots = u_m = 1,$$

$$u_{j+p} = \frac{1}{p} \sum_{i=j}^{j+p-1} \bar{u}_i \qquad j = 1, \ldots, n-p. \tag{2.13}$$

Now the only unknown in eq.(2.10) is $\mathbf{P}_i$. And $\mathbf{P}_i$ is solvable because there are $n+1$ equations and $n+1$ unknowns in eq.(2.10).

Now it comes to the surface interpolation. Given a set of $(n+1) \times (m+1)$ data points $\mathbf{Q}_{k,l}$, $k = 0, \ldots, n$ and $l = 0, \ldots, m$, a (p, q)th-degree B-spline surface interpolating these points is constructed by

$$\mathbf{Q}_{k,l} = \mathbf{S}(\bar{u}_k, \bar{v}_l) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(\bar{u}_k) N_{j,q}(\bar{v}_l) \mathbf{P}_{i,j}. \tag{2.14}$$

When using eq.(2.11) to compute $\bar{u}_k$, we can get m different values, which are $\bar{u}_k^l$, $l = 0, \ldots, m$. So it is decided to use the average

$$\bar{u}_k = \frac{1}{m+1} \sum_{l=0}^{m} \bar{u}_k^l \qquad k = 0, \ldots, n. \tag{2.15}$$

How to compute $\bar{v}_l$ is analogous. Again, in eq.(2.14) the only unknown is $\mathbf{P}_{i,j}$ which can be solved in a $((n+1) \times (m+1)) \times ((n+1) \times (m+1))$ linear system.

2.3.4 Projection

To project a point $\mathbf{P} = (x, y, z)$ onto a B-spline surface $\mathbf{S}(u, v)$, we need to obtain a pair of parameters $(\hat{u}, \hat{v})$, where the minimum distance between $\mathbf{P}$ and $\mathbf{S}(u, v)$ occurs [6]. Newton iteration can be applied to solve the problem and then $\mathbf{S}(\hat{u}, \hat{v})$ is the projection of $\mathbf{P}$.

Define the vector function

$$\mathbf{r}(u, v) = \mathbf{S}(u, v) - \mathbf{P}. \tag{2.16}$$

To minimize $\mathbf{r}(u, v)$, we must solve

$$\begin{aligned} f(u, v) &= \mathbf{r}(u, v) \cdot \mathbf{S}_u(u, v) = 0, \\ g(u, v) &= \mathbf{r}(u, v) \cdot \mathbf{S}_v(u, v) = 0. \end{aligned} \tag{2.17}$$

Let

$$\delta_i = \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = \begin{bmatrix} u_{i+1} - u_i \\ v_{i+1} - v_i \end{bmatrix},$$

$$J_i = \begin{bmatrix} f_u & f_v \\ g_u & g_v \end{bmatrix} = \begin{bmatrix} \|\mathbf{S}_u\|^2 + \mathbf{r} \cdot \mathbf{S}_{uu} & \mathbf{S}_u \cdot \mathbf{S}_v + \mathbf{r} \cdot \mathbf{S}_{uv} \\ \mathbf{S}_u \cdot \mathbf{S}_v + \mathbf{r} \cdot \mathbf{S}_{vu} & \|\mathbf{S}_v\|^2 + \mathbf{r} \cdot \mathbf{S}_{vv} \end{bmatrix},$$

$$\kappa_i = - \begin{bmatrix} f(u_i, v_i) \\ g(u_i, v_i) \end{bmatrix},$$

where all the functions in the matrix J_i are evaluated at (u_i, v_i). At the ith iteration, we must solve the 2×2 equation system

$$J_i \delta_i = \kappa_i \tag{2.18}$$

to compute

$$\begin{aligned} u_{i+1} &= \Delta u + u_i, \\ v_{i+1} &= \Delta v + v_i. \end{aligned} \tag{2.19}$$

To start the iteration, we need a first guess of the projection. In this thesis, we use the nearest neighbor of $\mathbf{P}$ as the first guess, and $\mathbf{S}(u_0, v_0)$ is the nearest neighbor. u_0 and v_0 are already computed in the interpolation process. The iteration of computing $(\hat{u}, \hat{v})$ does the following steps:

1. Check the condition of *point coincidence*

   $$\|\mathbf{S}(u_i, v_i) - \mathbf{P}\| \leq \epsilon_1.$$

   If it is true, return (u_i, v_i).

2. Check the condition of *zero cosine*

   $$\frac{|\mathbf{S}_u(u_i, v_i) \cdot (\mathbf{S}(u_i, v_i) - \mathbf{P})|}{\|\mathbf{S}_u(u_i, v_i)\| \|\mathbf{S}(u_i, v_i) - \mathbf{P}\|} \leq \epsilon_2 \qquad \frac{|\mathbf{S}_v(u_i, v_i) \cdot (\mathbf{S}(u_i, v_i) - \mathbf{P})|}{\|\mathbf{S}_v(u_i, v_i)\| \|\mathbf{S}(u_i, v_i) - \mathbf{P}\|} \leq \epsilon_2.$$

   If it is true, return (u_i, v_i).

3. Compute (u_{i+1}, v_{i+1}) by eq.(2.18) and eq.(2.19).

4. Modify (u_{i+1}, v_{i+1}) if they are out of the interval $[0, 1]$ as

   $$\begin{aligned} &\text{if } (u_{i+1} > 1) && u_{i+1} = 1, \\ &\text{if } (u_{i+1} < 0) && u_{i+1} = 0, \\ &\text{if } (v_{i+1} > 1) && v_{i+1} = 1, \\ &\text{if } (v_{i+1} < 0) && v_{i+1} = 0. \end{aligned}$$

5. Check the condition of convergence

   $$\|(u_{i+1} - u_i)\mathbf{S}_u(u_i, v_i) + (v_{i+1} - v_i)\mathbf{S}_v(u_i, v_i)\| \leq \epsilon_1.$$

   If it is true, return (u_{i+1}, v_{i+1}), if not, do the next iteration.

2.3.5 Complexity

Because the number of points in a NURBS surface patch is usually much smaller than N_A, so the complexity of NURBS surface interpolation and doing the projection can be ignored. Then the total complexity of NURBS surface interpolation mapping is the same as barycentric interpolation mapping and nearest neighbor mapping which is $O(N_A \cdot \log N_A) + O(N_B \cdot \log N_A)$ or $O(N \cdot \log N)$ when $N_A \approx N_B = N$.

2.4 RBF Interpolation Mapping

Radial basis function (RBF) interpolation is one of the mostly used method in interpolation of multidimensional scattered data. This method assumes that there is a function with respect to location describes the data field in a certain region. The data points (points with given data values) in $\mathbf{A}$ can be used to construct this function. Then by inputting an arbitrary location, the function value on this location is directly computed.

The radial basis function $\phi(r)$ is a function of distance with $r = \|\mathbf{x} - \mathbf{x_j}\|$, and $\mathbf{x}_j$ is the position vector of point j in $\mathbf{A}$, $\mathbf{x}$ is the position vector of an arbitrary point in space. RBF interpolation assumes that the data value $y(\mathbf{x})$ can be approximated by a linear combination of the ϕ's by

$$y(\mathbf{x}) = \sum_{i=1}^{n} w_i \phi(\|\mathbf{x} - \mathbf{x}_i\|) \tag{2.20}$$

where w_i's are the unknown weights [4]. Because the data values on points in $\mathbf{A}$ are already known, then we have n (which represents the number of points in $\mathbf{A}$) equations as

$$y(\mathbf{x}_j) = \sum_{i=1}^{n} w_i \phi(\|\mathbf{x}_j - \mathbf{x}_i\|) \qquad (j = 1, 2 \dots n) \tag{2.21}$$

which formulate a linear equation system

$$\begin{pmatrix} \phi(\|\mathbf{x}_1 - \mathbf{x}_1\|) & \cdots & \phi(\|\mathbf{x}_1 - \mathbf{x}_n\|) \\ \vdots & \ddots & \vdots \\ \phi(\|\mathbf{x}_n - \mathbf{x}_1\|) & \cdots & \phi(\|\mathbf{x}_n - \mathbf{x}_n\|) \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} y(\mathbf{x}_1) \\ \vdots \\ y(\mathbf{x}_n) \end{pmatrix}, \tag{2.22}$$

and the solution of the equation system are the weights. After the weights are solved, then given an arbitrary location in space the data value can be calculated by eq.(2.20).

The procedure of RBF interpolation mapping is that, first construct the data field function $y(\mathbf{x})$ by computing the weights in eq.(2.20), then substitute $\mathbf{x}$ by the position of the query point to compute the data value on it.

2.4.1 Radial Basis Functions

Radial basis functions can be divided into two groups, which are global RBF and compactly supported RBF. Four widely used types of global RBF's are listed below [4]:

- Multiquadric:
$$\phi(r) = (r^2 + {r_0}^2)^{1/2}. \tag{2.23}$$
- Inverse multiquadric:
$$\phi(r) = (r^2 + {r_0}^2)^{-1/2}. \tag{2.24}$$
- Thin-plate spline:
$$\phi(r) = r^2 \log(r/r_0). \tag{2.25}$$

- Gaussian:

$$\phi(r) = \exp(-\frac{1}{2}r^2/{r_0}^2). \tag{2.26}$$

And a famous compactly supported RBF is Wendland's function [5] as

$$\phi(r) = (1 - r/r_0)_+^4(4r/r_0 + 1), \tag{2.27}$$

where the subscript + means only the positive values are taken into account. It is easy to see when $r \leq r_0$, $\phi(r)$ becomes 0, so r_0 is called the support of the RBF.

The parameter r_0 in all RBF's has an influence on the accuracy and solvability of the interpolation. For example, if $r_0 \to \infty$, all types of RBF's except the thin-plate spline will become independent of r, and the matrix of the equation system becomes singular. The compactly supported RBF gives a rise to the non-singularity because there are more zeros in the matrix of the equation system. However, the author cannot find a literature that presents the laws on how to set r_0, so this will become a trouble in analyzing the performance of RBF interpolation mapping in Chapter 4.

2.4.2 Complexity

When computing the weights, a $N_A \times N_A$ linear equation system in eq.(2.22) is solved once. In this thesis, LU-decomposition is used for solving large linear equation systems. The complexity of LU-decomposition is $O({N_A}^3)$. In addition, for each point in $\mathbf{B}$, the function value is computed by a vector production (see eq.(2.20)) of complexity $O(N_A)$. So the total complexity of RBF interpolation mapping is $O({N_A}^3 + N_B \cdot N_A)$, or $O(N^3)$ when $N_A \approx N_B = N$.

2.5 Kriging Mapping

Kriging method is a spatial regression technique named for South African mining engineer D.G. Krige. Given a set of locations with data values in a certain region, kriging method can be used to estimate the data values at particular locations in the region.

2.5.1 Kriging

Here a brief introduction to the kriging method will be introduced. The knowledge is obtained mainly from the book [7].

Assume in a certain region there are a set of points $\{\mathbf{x}_i, i = 1, \ldots, n\}$ with data value $y(\mathbf{x}_i)$ on each point, and we want to estimate the data value on a new point $\mathbf{x}_0$. The estimation by Kriging method is a linear combination of the data values on all other points as

$$y^*(\mathbf{x}_0) = \sum_{i=1}^{n} w_i y(\mathbf{x}_i). \tag{2.28}$$

The weights w_i should sum up to one because if all data values are equal to a constant, the estimated value should be equal to this constant.

In regression, the goal is to minimize the variance between the estimated value and the real value. The estimation variance of $y(\mathbf{x}_0)$ is

$$\begin{aligned}\sigma^2 &= E[(y^*(\mathbf{x}_0) - y(\mathbf{x}_0))^2] \\ &= -\gamma(\mathbf{x}_0 - \mathbf{x}_0) - \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \gamma(\mathbf{x}_i - \mathbf{x}_j) \\ &+ 2\sum_{i=1}^{n} w_i \gamma(\mathbf{x}_i - \mathbf{x}_0) - 2\mu(\sum_{i=1}^{n} w_i - 1),\end{aligned} \tag{2.29}$$

where $\gamma(h)$ is called *variogram* which will be discussed later, and μ is the Lagrange parameter. The variance is minimal when

$$\frac{\partial \sigma^2}{\partial w_i} = 0, \qquad \text{for } i = 1, \ldots, n. \tag{2.30}$$

Then we have

$$\sum_{j=1}^{n} w_j \gamma(\mathbf{x}_i - \mathbf{x}_j) + \mu = \gamma(\mathbf{x}_i - \mathbf{x}_0), \qquad \text{for } i = 1, \ldots, n \tag{2.31}$$

$$\sum_{j=1}^{n} w_j = 1, \tag{2.32}$$

which can be rewritten in matrix form

$$\begin{pmatrix} \gamma(\mathbf{x}_1 - \mathbf{x}_1) & \cdots & \gamma(\mathbf{x}_1 - \mathbf{x}_n) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(\mathbf{x}_n - \mathbf{x}_1) & \cdots & \gamma(\mathbf{x}_n - \mathbf{x}_n) & 1 \\ 1 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_n \\ \mu \end{pmatrix} = \begin{pmatrix} \gamma(\mathbf{x}_1 - \mathbf{x}_0) \\ \vdots \\ \gamma(\mathbf{x}_n - \mathbf{x}_0) \\ 1 \end{pmatrix}. \tag{2.33}$$

By solving the linear equation system, we can obtain the weights. Note that this method is also an interpolation method because if $\mathbf{x}_0 = \mathbf{x}_k$, then $y(\mathbf{x}_0) = y(\mathbf{x}_k)$. The reason is easy, when $\mathbf{x}_0 = \mathbf{x}_k$, the solution will be $w_k = 1$ with all other weights equal to 0, since one column in the left hand side matrix is the same as the right hand side.

2.5.2 Variogram

The detail of variogram is beyond the scope of this thesis, you can refer to [7] to get the background knowledge. It is an estimate of the mean square of the function $y(\mathbf{x})$ [4], by

$$\gamma(\mathbf{r}) \sim \frac{1}{2}(y(\mathbf{x} + \mathbf{r}) - y(\mathbf{x}))^2. \tag{2.34}$$

$\gamma(\mathbf{r})$ is usually simplified by $\gamma(r)$ where $r = \|\mathbf{r}\|$ [4].

In [4], three variogram functions are introduced, they are

- Power:
$$\gamma(r) = ar^b. \tag{2.35}$$
- Exponential:
$$\gamma(r) = b(1 - \exp(-r/a)). \tag{2.36}$$
- Spherical:
$$\gamma(r) = \begin{cases} b(\frac{3}{2}\frac{r}{a} - \frac{1}{2}\frac{r^3}{a^3}) & 0 \le r \le a \\ b & a \le r \end{cases} \tag{2.37}$$

There are two parameters in each variogram function above, which determines the shape of the function. In this thesis, how to set the parameters is not investigated, but in [4], an algorithm for setting parameters of *power* variogram is given. The parameter b is set to 1.5, and a is computed by

$$a = \frac{1}{2}\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(\|\mathbf{x}_i - \mathbf{x}_j\|)^{b/2}\sqrt{y(\mathbf{x}_i) - y(\mathbf{x}_j)}}{\sum_{i=1}^{n}\sum_{j=1}^{n}\sqrt{(\|\mathbf{x}_i - \mathbf{x}_j\|)^{b/2}}}. \tag{2.38}$$

2.5.3 Complexity

We can combine eq.(2.28) and eq.(2.33) together as

$$y(\mathbf{x}_0) = \mathbf{V}_0 \cdot \mathbf{V}^{-1} \cdot \mathbf{Y}, \tag{2.39}$$

where $\mathbf{V}_0$ is the right hand side in eq.(2.33), $\mathbf{V}$ is the left hand matrix in eq.(2.33), and $\mathbf{Y} = (y(\mathbf{x}_1), \ldots, y(\mathbf{x}_n), 0)^T$.

The straightforward way to compute $y(\mathbf{x}_0)$ is to solve $\mathbf{V}_0 \cdot \mathbf{V}^{-1}$ firstly, and then use the result (which is the vector of weights) times $\mathbf{Y}$. So we need to do LU-decomposition for $\mathbf{V}$ once, which has the complexity $O({N_A}^3)$, and for each query point $\mathbf{x}_0$ we need to do back substitution for $\mathbf{V}_0$ once, which has a complexity $O(N_B \cdot {N_A}^2)$. Then the overall complexity is $O({N_A}^3) + O(N_B \cdot {N_A}^2)$. [4] indicates a better way, which we can solve $\mathbf{V}^{-1} \cdot \mathbf{Y}$ firstly, in the equation system

$$\mathbf{V} \cdot (\mathbf{V}^{-1} \cdot \mathbf{Y}) = \mathbf{Y}, \tag{2.40}$$

and then do a vector production. So we do LU-decomposition and a back substitution only once, which has the complexity $O({N_A}^3)+O({N_A}^2) = O({N_A}^3)$. For each query point, we do a vector product, so the complexity is $O(N_B \cdot N_A)$. So the overall complexity is $O({N_A}^3 + O(N_B \cdot N_A))$, and simply $O(N^3)$ when $N_A \approx N_B = N$.

2.6 A Failed Idea

During the investigation of possible mapping methods, one could think of using some higher order polynomial to do the mapping. The higher order polynomial is

$$f(x, y, z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9, \tag{2.41}$$

where $f(x, y, z)$ represents the data value on the point with coordinates (x, y, z), and $a_0, \ldots, a_9$ are the coefficients. If the coefficients are known, then the data value on the query point can be obtained by eq.(2.41). So we can first search ten nearest neighboring points of the query point, and by inputing the data values and coordinates of these points into eq.(2.41), the coefficients can be solved in a 10×10 linear equation system. However, the implementation of the method failed because the coefficient matrix in the equation system is often rank deficient, and this method gives big amount of error after solving the rank deficiency.

3 Implementation

The coding of the c++ program *MeshMapping* is the major job in this work. The program is able to read the mesh file and the data file, do the mapping using the methods introduced in Chapter 2, and output the result. The functionality of the program can be grouped into three parts:

1. Read and write the files of mesh and data. The files are formatted by some visualization software, so that the mapping results can be visualized.

2. Map the data from one mesh to the other mesh. All the mapping methods in Chapter 2 are implemented.

3. Read the *setting* file to run the program. The setting file contains some options to offer a good flexibility to avoid compiling the program each time if the test case is changed.

In the following, the implementation of above functionalities including the related code will be explained.

3.1 Reading and Writing the Mesh File and the Data File

The software GiD [9] is a pre and postprocessing tool, which is used for geometrical modeling and visualization in this work. In GiD, we can create a surface and mesh it. The mesh can be exported as a so-called *GiD mesh file*, which is with file extension `.msh`. Because GiD could not do analysis or computation, it is not a good choice to generate analysis data. But it provides a special file format for the data, which is called *GiD post result file*, with file extension `.res`. With a `.msh` file and a `.res` file, GiD can do visualization of the data.

3.1.1 File Format of `.msh` and `.res`

A typical `.msh` file is given below. The first line tells us some basic information of the mesh. The string after the keyword "MESH" is the name of the mesh. After that is the dimension of the mesh which equals 3 here, so each node in the mesh has three coordinates (x, y, z). ("Node" is a substitute of "point", because it is more usual to speak "node" in the FEM community.) The element type is 4 nodes' quadrilateral, so each element consists 4 nodes, and these nodes construct a quadrilateral. After the first

line it comes the *coordinates* block where the index and the coordinates of each node is listed line by line. Then it comes the *elements* block where the index and consisting nodes of each element is listed line by line. One comment is that, the example `.msh` file contains only one mesh, but there could be more than one meshes in each `.msh` file. However, the program *MeshMapping* specifies that there should be only one mesh in each `.msh` file, i.e. the mesh with data and the mesh without data should be in separate files.

```
MESH "example" dimension 3 ElemType Quadrilateral Nnode 4
Coordinates
    1               0               0               0
    2               0               0             0.5
    3             0.5               0               0
    4             0.5               0             0.5
    5               0               0               1
    6               1               0               0
    7               1               0             0.5
    8             0.5               0               1
    9               1               0               1
end coordinates

Elements
       1       3       4       2       1
       2       6       7       4       3
       3       4       8       5       2
       4       7       9       8       4
end elements
```

A `.res` file contains the data on some mesh, which is called "result" by GiD. Since there can be different types of data (e.g. pressure, displacements, ...) on the same mesh, so a `.res` file can have multiple results. In the `.res` file below, the displacements and the pressures belong to the same mesh in the `.msh` file above. The header line "GiD Post Results File 1.0" is necessary for identification. The first line of each result provides the basic information. After the keyword "Result", it comes sequentially the result name, the analysis name, the time step number, the type of the data, and the location of the data. The result name, the analysis name and the time step number are used for the purpose of showing post processing information in GiD. The type of data corresponds to how many data values are contained on each node. So if the type of the data is *Vector*, there are *three* data values on each node. If the type is *Scalar*, then there is only *one* data value on each node. The location of the data values in both results is "OnNodes". Another option of the location of the data is *OnGaussPoints*, which means the data locate on the Gauss points of each element instead of on the nodes. But it is not needed for our mapping usage. In each result there is a block of "values", which contains the data values on each node.

```
GiD Post Results File 1.0
```

```
Result "Displacements" "Load Analysis"  1  Vector OnNodes
Values
    1    0.0   0.0   0.0
    2   -0.1   0.1   0.5
    3    0.0   0.0   0.8
    4   -0.04  0.04  1.0
    5   -0.05  0.05  0.7
    6    0.0   0.0   0.0
    7   -0.04 -0.04  1.0
    8    0.0   0.0   1.2
    9   -0.1  -0.1   0.5
End Values

Result "Pressure" "Load Analysis"  1 Scalar OnNodes
Values
    1    0.0
    2   -0.1
    3    0.0
    4   -0.04
    5   -0.05
    6    0.0
    7   -0.04
    8    0.0
    9   -0.1
End Values
```

3.1.2 Classes of the Data

When doing file I/O with `.msh` and `.res` files, there must be some data structures in the program to maintain the data in those files. When defining these data structures, the idea is to include all the information in a file even if some are not necessary for mapping, so that the program can be easily extended in the future (e.g. we want to map the data on Gauss points).

The class which maintains the data in a `.msh` file is MMeshInventory. It has only one field, a vector of MMesh*. This vector contains all the meshes in a `.msh` file.

```
class MMeshInventory {
public:
  MMeshInventory(vector<MMesh*> *vecMesh) : _vecMesh(vecMesh) {
  }
  vector<MMesh*> *getMeshes() {return _vecMesh;}
  ...

private:
  vector<MMesh*> *_vecMesh;
};
```

The class MMesh maintains a mesh in a `.msh` file. The first four fields correspond to the information in the first line of a mesh in the `.msh` file. The vector of MNode* corresponds to the coordinates block and the vector of MElement* corresponds to the elements block of the mesh.

```
class MMesh {
public:
  MMesh(
    string name, string elementType, int dimension,
    int nodesPerElem, vector<MNode*> *nodes, vector<MElement*> *elements
  ): _name(name), _elementType(elementType), _dimension(dimension),
    _nodesPerElem(nodesPerElem), _nodes(nodes), _elements(elements) {
  }
  vector<MNode*> *getNodes() const {return _nodes;}
  vector<MElement*> *getElements() const {return _elements;}
  ...

private:
  string _name;
  string _elementType;
  int _dimension;
  int _nodesPerElem;
  vector<MNode*> *_nodes;
  vector<MElement*> *_elements;
};
```

The class MNode corresponds to a line in the coordinates block. It has two fields. _id is the index of the node, and _coor is a pointer to a dynamic array of coordinates.

```
class MNode {
public:
  MNode(int id, double *coor) : _id(id), _coor(coor) {
  }
  int getId() const {return _id;}
  double* getCoordinate() const {return _coor;}
  ...

private:
  int _id;
  double *_coor;
};
```

The class MElement corresponds to a line in the elements block. It has two fields. _id is the index of the element, and _nodeIds is a pointer to a dynamic array which contains the indexes of the nodes that construct this element.

```
class MElement {
public:
```

```
  MElement(int id, int *nodeIds) :_id(id), _nodeIds(nodeIds) {
  }
  int getId() const {return _id;}
  int *getNodeIds() const {return _nodeIds;}
  ...

private:
  int _id;
  int *_nodeIds;
  ...
};
```

The class which maintains the data in a `.res` file is MResultInventory. The vector of MGaussPointSetting* contains all the Gauss point settings in a `.res` file. However, it is not used in the mapping since only the data on the nodes are mapped. The vector of MResult* contains all the results in a `.res` file.

```
class MResultInventory {
public:
  MResultInventory(
    vector<MGaussPointSetting*> *vecGPS, vector<MResult*> *vecResult
  ): _vecGPS(vecGPS), _vecResult(vecResult) {
  }
  vector<MResult*> *getResults() { return _vecResult;}
  ...

private:
  vector<MGaussPointSetting*> *_vecGPS;
  vector<MResult*> *_vecResult;
};
```

The class MResult maintains a result in a `.res` file. It has seven fields. The first six fields correspond to the information in the first line of the result (again the last one is the name of the Gauss point setting but it is not useful for mapping). The vector of MValue* corresponds to the values block in a result.

```
class MResult {
public:
  MResult(
    string name, string analysisName, int stepNum, string type,
    bool onNodes, string GPSName, vector<MValue*> *vecValue
  ): _name(name), _analysisName(analysisName), _stepNum(stepNum),
    _type(type), _onNodes(onNodes), _GPSName(GPSName),
    _vecValue(vecValue) {
  }
  vector<MValue*> *getValues() const { return _vecValue; }
  ...
```

```
private:
  string _name;
  string _analysisName;
  int _stepNum;
  string _type;
  bool _onNodes;
  string _GPSName;
  vector<MValue*> *_vecValue;
};
```

The class MValue corresponds to a line in the values block. It has two fields. _id is the index of the node to which the values belongs, and _value is a pointer to a dynamic array which contains the the data values on the node.

```
class MValue {
public:
  MValue(int id, double *value) : _id(id),  _value(value){
  }
  int getId() const {return _id;}
  double *getValue() const {return _value;}
  ...

private:
  int _id;
  double *_value;
  ...
};
```

3.1.3 Class FileIO

The class FileIO is in charge of reading and writing the `.msh` and `.res` files. It has four *static* methods. readDotMsh reads a `.msh` file and constructs an object of MMeshInventory by the content of the file, and the pointer to the object is returned at the end of this method. writeDotMsh extracts the data in MI and writes them to a `.msh` file with name fileName. readDotRes and writeDotRes do analogous jobs.

```
class FileIO {
public:
  static MMeshInventory *readDotMsh(string fileName);
  static MResultInventory *readDotRes(string fileName);
  static void writeDotMsh(MMeshInventory *MI, string fileName);
  static void writeDotRes(MResultInventory *RI, string fileName);
  ...
};
```

3.2 Doing the Mapping

In the program *MeshMapping* , each mapping method is encapsulated in a separate class. All these classes extend the same super class AbstractMapping. In this class, _meshFrom is the mesh which has given results _resultsFrom. _meshTo is the mesh to which we want to map the results. doMapping is an abstract method, each child class of AbstractMapping should implement this method to perform the mapping.

```
class AbstractMapping {
public:
  AbstractMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom
  ): _meshFrom(meshFrom), _meshTo(meshTo), _resultsFrom(resultsFrom) {
  }
  virtual vector<MResult*> *doMapping() = 0;
  ...

protected:
  MMesh *_meshFrom;
  MMesh *_meshTo;
  vector<MResult*> *_resultsFrom;
  ...
};
```

3.2.1 Implementation of Nearest Neighbor Mapping

The class NearestNeighborMapping implements nearest neighbor mapping.

```
class NearestNeighborMapping : AbstractMapping {
public:
  NearestNeighborMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom
  ): AbstractMapping(meshFrom, meshTo, resultsFrom) {
  }
  vector<MResult*> *doMapping();
  ...
};
```

The algorithm of the method doMapping is as following:

Initialize vector<MResult*> *resultsTo which will maintain the data on _meshTo
Build a kd-tree with all nodes in _meshFrom
for node i **in** _meshTo
  Use the kd-tree to find the nearest node j of i
  Assign the data values of i to be the same as those of j

Put the data values on i in resultsTo
end for
return resultsTo

3.2.2 Implementation of Barycentric Interpolation Mapping

The class ProjectionMapping implements barycentric interpolation mapping.

```
class ProjectionMapping: public AbstractMapping {
public:
  ProjectionMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom
  ): AbstractMapping(meshFrom, meshTo, resultsFrom) {
  }
  vector<MResult*> *doMapping();
  ...
};
```

The algorithm of the method doMapping is as following:

Initialize vector<MResult*> *resultsTo which will maintain the data on _meshTo
Build a kd-tree with all nodes in _meshFrom
for node i **in** _meshTo
  Use the kd-tree to find the nearest *three* nodes of i
  while (the three nodes locate in the same line)
    Find the next nearest node, which is used to replace the third nearest node
  end while
  Compute the projection of i on the triangle using eq.(2.1)
  Do barycentric interpolation to compute w_1, w_2, w_3 using eq.(2.3)
  Use the data values on the three neighboring nodes to compute the data values on i using eq.(2.2)
  Put the data values on i in resultsTo
end for
return resultsTo

The verification of whether three nodes locate in the same line, the projection, and the barycentric interpolation are performed in the class MComputation. The equation system in eq.(2.1) is solved by Gauss elimination in MComputation. The algorithms of these methods are not complex so they will not be explained here.

```
class MComputation {
public:
  static void doProjection(
```

```
    double *coor1, double *coor2, double *coor3,
    double *coor0, double *coor4
  );
  static void doBarycentricInterpolation(
    double *coor1, double *coor2, double *coor3,
    double *coor4, double *weights
  );
  static void doGaussElimination(
    double **LHS, double *RHS, const int num, double *x
  );
  static bool verifyNonLinear(
    double *coor1, double *coor2, double *coor3
  );
  ...
}
```

3.2.3 Implementaion of NURBS Surface Interpolation Mapping

The class NurbsSurface implements all NURBS-related algorithms used in the mapping. There are two **public** methods in NurbsSurface, the constructor and the method projection. The constructor builds a NURBS surface interpolating the inputting points, and projection computes the projection of a point on a NURBS surface. All methods in NurbsSurface use the algorithms provided by [6] (most of which are written in C-like code), and the comments before each method indicate the place of the algorithm in [6].

```
class NurbsSurface {
public:
  /* see "The NURBS Book", P380, A9.4 */
  NurbsSurface(double ***Q, int num, int n, int m, int p, int q);

  /* see "The NURBS Book", P232~P234 */
  void projection(double *P, int *guess, double *Prj);

  ...

private:
  double ***_P;
  int _num;
  int _n;
  int _m;
  int _p;
  int _q;
  double *_U;
  double *_V;
  double *_uk;
  double *_vl;
```

```
...

/* see "The NURBS Book", P377, A9.3 */
void surfMeshParams(double ***Q);

/* see "The NURBS Book", P68, A2.1 */
int findSpan(int n, int p, double u, double *U);

/* see "The NURBS Book", P70, A2.2 */
void basisFuns(int i, double u, int p, double *U, double *N);

/* see "The NURBS Book", P99 */
void allBasisFuns(int i, double u, int p, double *U, double **N);

/* see "The NURBS Book", P369, A9.1 */
void globalCurveInterp(
  int n, double **Q, double *uk, int r,
  int p, double *U, double **P
);

/* see "The NURBS Book", P365, eq(9.8) */
void knotVector(int n, int p, double *uk, double *U);

/* see "The NURBS Book", P99, A3.3 */
void curveDerivCpts(
  int p, double *U, double **P, int d,
  int r1, int r2, double ***PK
);

/* see "The NURBS Book", P99, A3.4 */
void curveDerivsAlg2(
  int n, int p, double *U, double **P,
  double u, int d, double **CK
);

/* see "The NURBS Book", P114, A3.7 */
void surfaceDerivCpts(
  int d, int r1, int r2, int s1,
  int s2, double *****PKL
);

/* see "The NURBS Book", P115, A3.8 */
void surfaceDerivsAlg2(double u, double v, int d, double ***SKL);

/* see "The NURBS Book", P103, A3.5 */
void surfacePoint(double u, double v, double *S);
```

```
  ...
};
```

In this class, _P is an array of all control points of the NURBS surface, as in eq.(2.8). _num is the number of data values on each point, e.g. if a point has coordinates (x, y, z) and has pressure p and displacements $(x - disp, y - disp, z - disp)$ on it, then _num= 7. If _n=n and _m=m, then there are $(n + 1) \times (m + 1)$ control points. _p and _q are the degree of B-spline functions in eq.(2.8). _U and _V are two knot vectors. _uk and _vl are the parameter values where the points in a patch are interpolated by the B-spline surface(see eq.(2.14)).

The input arguments of the constructor include the basic setting parameters of the NURBS surface as well as the points in a patch which are represented by Q. The constructor should construct a NURBS surface that interpolates all points in Q. The algorithm inside the constructor is as following:

Assign the values of _num, _n, _m, _p and _q by the input arguments
Call the method surfMeshParams to calculate _uk and _vl using eq.(2.11)
Call the method knotVector to calculate knot vectors _U and _V using eq.(2.13)
Call the method globalCurveInterp to compute the control points of the NURBS surface that interpolates all the points in Q by solving eq.(2.14).

There are two input arguments and one output argument of the method projection. P is the point that is going to be projected on the NURBS surface. To use Newton iteration to obtain the projection as introduced in Section 2.3.4, there must be an initial guess to start Newton iteration. It is obvious that the nearest neighbor is a good guess. The argument guess is an array containing two indexes i and j. When substitute u and v in eq.(2.8) by _uk[i] and _vl[j], we can get a point which is actually the nearest neighbor. The output argument Prj is the projection point. Section 2.3.4 has already explained the algorithm of projection. There are two important methods that are called in projection, which are surfacePoint and surfaceDerivsAlg2. surfacePoint computes a point on the NURBS surface using eq.(2.8). surfaceDerivsAlg2 computes the up to a certain order derivatives of a point. Other methods in NurbsSurface will not be introduced, they are auxiliary methods used by the methods explained above.

The class NurbsInterpMapping is in charge of performing NURBS surface interpolation mapping. _patchDim is the size of the patch, i.e. if _patchDim=d, then the patch is constructed by $d \times d$ points. _degree is the degree of the B-spline functions in eq.(2.8) (i.e. $p = q =$_degree).

```
class NurbsInterpMapping : AbstractMapping {
public:
  NurbsInterpMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom,
    int patchDim, int degree
```

```
    ): AbstractMapping(meshFrom, meshTo, resultsFrom) {
      _patchDim = patchDim;
      _degree = degree;
    }
    vector<MResult*> *doMapping();
    ...

  private:
    int _patchDim;
    int _degree;
    ...
  };
```

The algorithm of the method doMapping is as following:

if (the element type of _meshFrom is not four points' quadrilateral)
  return NULL
*Renumbering the indexes of the nodes in _meshFrom
Initialize vector<MResult*> *resultsTo which will maintain the data on _meshTo
Build a kd-tree with all nodes in _meshFrom
for node i **in** _meshTo
  Use the kd-tree to find the nearest node of i
  *Define the NURBS surface patch
  Build a NURBS surface interpolating all nodes in the patch by creating a new instance ns of the class NurbsSurface
  Compute the projection of i on the NURBS surface by calling the method projection of ns
  Assign the data values on i to be the same as those on the projection
  Put the data values of i in resultsTo
end for
return resultsTo

The processes marked by * need some more explanation. When creating a four nodes' quadrilateral mesh in GiD, the nodes in the mesh are not numbered regularly while the elements are numbered regularly. So in doMapping, the indexes of all nodes will be renumbered according to the numbering of the elements (see Figure 3.1). The algorithm of the renumbering is not difficult, so only a hint is given here. The algorithm first determines m and n, if _meshFrom has $m \times n$ elements. Then the indexes of the eight surrounding elements of a special element (if there exist) are known, and now we have enough information to renumber the indexes of the nodes. For example, the node in element no.1 which is not in element no.2 and element no.(m+1) has a new index 1 (see Figure 3.2). The new indexes of other nodes can be determined in an analogous way. So besides the requirement of *four nodes' quadrilateral*, there is a new requirement for the mesh with given data, that it should have regularly numbering elements. The reason

why the renumbering is performed is that, the new numbering of nodes provide an easy way to define the patch. For example, we know there are $(m+1) \times (n+1)$ nodes in _meshFrom, if a 3×3 points patch is desired, and if the index of the nearest neighbor is i, then the patch is defined by the nodes $i-(m+1)-1$, $i-(m+1)$, $i-(m+1)+1$, $i-1$, i, $i+1$, $i+(m+1)-1$, $i+(m+1)$ and $i+(m+1)+1$. But if i is near to the boundary of the mesh, there will be problem. The solution is to adjust i with some offsets, as $i = i + \text{offset1} + \text{offset2} \cdot (m+1)$. Another problem in defining the patch is that, if the patch has $d \times d$ nodes, then whether d is odd or even will demand different algorithms. If d is odd, then the patch is defined with the nearest neighbor located in the center. If d is even, then more neighbors should be searched to find the *nearest element*, and then the patch is defined with the nearest element located in the center. There are four candidates of the nearest element, each element has a node opposite to the nearest neighbor, so the nearest element is defined by the element whose node opposite to the nearest neighbor is closer to the query node. This is shown in Figure 3.3.

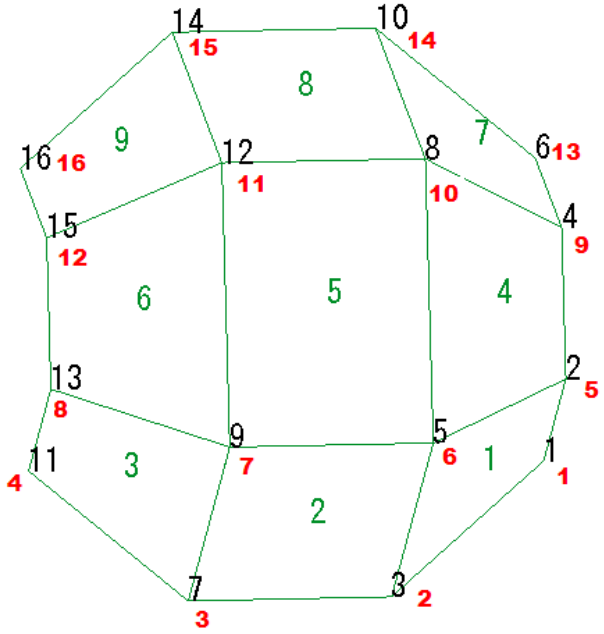


Figure 3.1: A mesh created in GiD. The green numbers are the indexes of the elements while the black numbers are the indexes of the nodes. These numbers are generated by GiD automatically. The red numbers are the expected numbering of nodes.

3.2.4 Implementation of RBF Interpolation Mapping

The class RadialBasisFuncInterpolation implements all the computation of RBF interpolation Mapping. _dim is the number of coordinates of all points. _n is the number of points in _ptsMatrix, which is an array of all the points $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ in eq.(2.20)). _fn

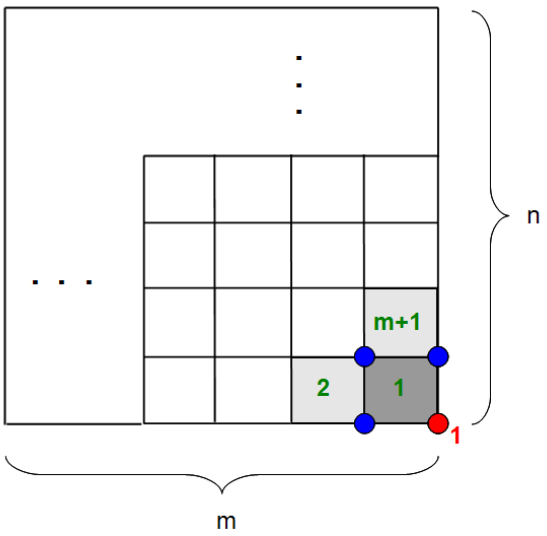


Figure 3.2: An illustration of how to determine the new indexes of nodes.

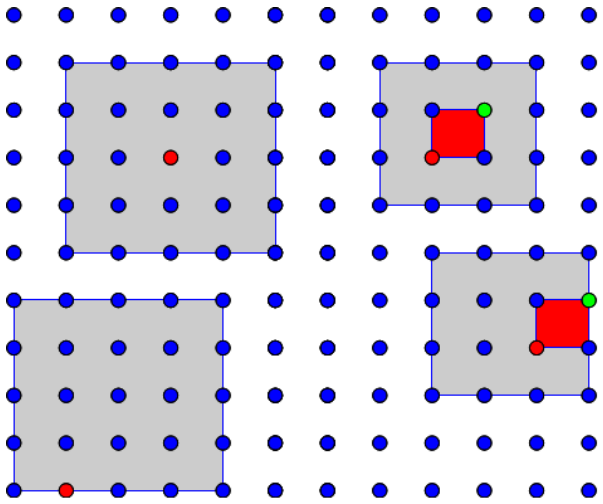


Figure 3.3: Defining the patch. The red point is the most nearest neighbor, the green point is closer to the query point than other three candidates so that the nearest element is obtained which is the red square.

is the RBF that is chosen ($\phi(r)$ in eq.(2.20)). The five types of RBF introduced in Section 2.4.1 are implemented in this class. The code of the RBF is quite simple so that it is skipped in the code part below. _lu is an object of the class LUDecomposition, which solves the linear equation system in eq.(2.22). LUDecomposition implements the algorithms of LU-decomposition introduced in [4]. LU-decomposition is a direct method for solving linear equation systems. Its advantage to Gauss elimination is that if there are more than one right hand side vectors corresponding to the same coefficient matrix, then the method performs decomposition (of complexity $O(N^3)$ which is the same as Gauss elimination) only once and do several substitutions (of complexity $O(N^2)$). Because there are usually multiple data values on a point, which derives multiple right hand side vectors corresponding to the same coefficient matrix in eq.(2.22), LU-decomposition fits here. The detail of LU-decomposition is beyond the scope of mapping, so it will not be explained in this thesis. _multWeights represents several groups of weights, each group corresponds to the weights in eq.(2.20). _num is the number of data values on each point, e.g. if a point has data values pressure p and displacements $(x-disp, y-disp, z-disp)$, then _num= 4. In the constructor of RadialBasisFuncInterpolation, the points and the type of RBF is passed in to build the coefficient matrix in eq.(2.22), then the matrix is used to create an instance of LUDecomposition, which LU-decomposes the matrix. In the method computeMultipleWeights, the data values on all points are passed in, where vals[i][j] is the jth value on the ith point. Each point has num data values, so num right hand vectors are built and _multWeights is solved by substitution. The method interpolateMultiple uses eq.(2.20) to solve all data values y on an arbitrary point p.

```
class RadialBasisFuncInterpolation {
public:
  RadialBasisFuncInterpolation(
    int dim, int n, double **ptsMatrix, RBFFuncType type
  );
  void computeMultipleWeights(double **vals, int num);
  void interpolateMultiple(double *p, double *y);
  ...

private:
  int _dim;
  int _n;
  double **_ptsMatrix;
  RBFFunc *_fn;
  LUDecomposition *_lu;
  double **_multWeights;
  int _num;
  ...
};
```

The class RBFInterpolationMapping performs RBF interpolation mapping. _type reprensents the type of RBF.

```
class RBFInterpolationMapping : AbstractMapping {
public:
  RBFInterpolationMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom,
    RadialBasisFuncInterpolation::RBFFuncType type
  ): AbstractMapping(meshFrom, meshTo, resultsFrom), _type(type) {
  }
  vector<MResult*> *doMapping();
  ...

private:
  RadialBasisFuncInterpolation::RBFFuncType _type;
  ...
};
```

The algorithm of the method doMapping is as following:

Initialize vector<MResult*> *resultsTo which will maintain the data on _meshTo
Create an instance rbfi of RadialBasisFuncInterpolation with all nodes in _meshFrom
Call the method computeMultipleWeights of rbfi by passing in data values on all points, then the weights in eq.(2.22) are solved.
for node i **in** _meshTo
  Call the method interpolateMultiple of rbfi by passing in the coordinates of i, to compute the data values on i by eq.(2.20).
  Put the data values on i in resultsTo
end for
return resultsTo

3.2.5 Implementation of Kriging Mapping

The class Kriging implements all the computation of Kriging mapping. _npt is the number of points in _ptsMatrix, which is an array of all points. _dim is the number of coordinates that each point has. _vgram is the variogram function that is chosen. Here only the *power* variogram is implemented because methods for setting the parameters in other variograms introduced in Section 2.5.2 are not found. The *power* variogram is implemented inside this class, but the code of it is omitted since it is quite simple. _lu is responsible for solving eq.(2.40) by LU-decomposition. _weights is the solution of eq.(2.40) (_weights is not the weights in eq.(2.35) which is already explained in Section 2.5.3). In the constructor, all the points ptsMatrix and the data values on these points vals are passed in, so that $\mathbf{V}$ and $\mathbf{Y}$ in eq.(2.40) are obtained, and the equation can be solved. In the method interpolate, an arbitrary point p is passed in to solve the data value on it using eq.(2.39).

```
class Kriging {
```

```
public:
  Kriging(
    int npt, int dim, double **ptsMatrix,
    double *vals, VariogramType type
  );
  double interpolate(double *p);
  ...

private:
  int _npt;
  int _dim;
  double ** _ptsMatrix;
  Variogram *_vgram;
  LUDecomposition *_lu;
  double *_weights;
  ...
};
```

The class KrigingMapping performs Kriging mapping. The field _type is the type of variogram functions.

```
class KrigingMapping : AbstractMapping {
public:
  KrigingMapping(
    MMesh *meshFrom, MMesh *meshTo, vector<MResult*> *resultsFrom,
    Kriging::VariogramType type = Kriging::POWER,
  ): AbstractMapping(meshFrom, meshTo, resultsFrom),_type(type) {
  }
  vector<MResult*> *doMapping();
  ...

private:
  Kriging::VariogramType _type;
  ...
};
```

The algorithm of the method doMapping is as following:

Initialize vector<MResult*> *resultsTo which will maintain the data on _meshTo
for $k <$ total number of data values
  Create an instance krig of the class Kriging with all nodes in _meshFrom and the kth data value on all points
  for node i **in** _meshTo
    Call the method interpolate of krig by passing in the coordinates of i, to compute the kth data value on i by eq.(2.39).
    Put the kth data value of i in resultsTo

```
  end for
end for
return resultsTo
```

3.2.6 Class MappingManager

The class MappingManager encapsulates all mapping methods. _MIFrom is the mesh which has results _RIFrom, and _MITo is the mesh whose results are going to be computed by mapping.

```
class MappingManager {
public:
  MappingManager(
    MMeshInventory *MIFrom,MMeshInventory *MITo,MResultInventory *RIFrom
  ): _MIFrom(MIFrom), _MITo(MITo), _RIFrom(RIFrom) {
  }
  MResultInventory *doNearestNeighborMapping();
  MResultInventory *doProjectionMapping();
  MResultInventory *doRBFInterpolationMapping(string type);
  MResultInventory *doKrigingMapping(string type);
  MResultInventory *doNurbsSurfaceMapping(int patchDim, int degree);
  ...

private:
  MMeshInventory *_MIFrom;
  MMeshInventory *_MITo;
  MResultInventory *_RIFrom;
};
```

In each method above, an instance of the class doing special mapping is created and the method doMapping of the instance is called. The output of doMapping will be used to create a new instance of MResultInventory, the pointer to which is returned at the end. The code of doProjectionMapping is put below to provide an example:

```
MResultInventory *MappingManager::doProjectionMapping() {
  MMesh *meshFrom = _MIFrom->getMeshes()->at(0);
  MMesh *meshTo = _MITo->getMeshes()->at(0);
  vector<MResult*> *resultsFrom = _RIFrom->getResults();
  ProjectionMapping *projMapping =
      new ProjectionMapping(meshFrom,meshTo,resultsFrom);
  vector<MResult*> *results = projMapping->doMapping();
  delete projMapping;
  if(results == NULL)
    return NULL;
```

```
    vector<MGaussPointSetting*> *vecGPS =
        new vector<MGaussPointSetting*>();
    return new MResultInventory(vecGPS,results);
  }
```

3.3 Run the Program

When the program is launched, it will first prompt to choose the running mode as

```
Choose running mode:  mapping or performance (m/p)?
```

If the mode "mapping" is chosen, then the program will read the file `setting.txt` to perform the mapping. The file `setting.txt` locates in the same directory as the executable program. A prototype of the file is listed below. The file has some instructions explaining itself. In the example, the program will map the data from the mesh in the file `plane40x40.msh` to the mesh in the file `plane10x10.msh` with all mapping methods. The data file is not provided by the user, so the program will automatically generate some data (e.g. $f(x, y, z) = \sin(x) + \sin(y) + \sin(z)$) to test the mapping. At the end, the program will generate a `.res` file for the result of each mapping method. The `.res` file and its corresponding `.msh` file can be read by GiD to visualize the mapping result. The class SettingParser is in charge of parsing the file, but it will not be explained here.

```
# The user input place is between '{' and '}'.
# The output of the program is written to the folder 'mapping_results'.
=========================================================================
1. Input data
# Please input the mesh file with known result after 'mesh from:',
#   and the mesh file without result after 'mesh to:'.
# For usual mapping purpose, type 'y' after 'result provided:'
#   and input the result file after 'result from:'.
# For testing purpose, input nothing after 'result provided:'.
  mesh from: {plane40x40.msh}
  mesh to: {plane10x10.msh}
  result provided: {}
  result from: {}
=========================================================================
2. Setting mapping methods

2.1 Nearest Neighbor Mapping
# Type 'y' after 'do:' to perform nearest neighbor mapping.
  do: {y}
-------------------------------------------------------------------------
2.2 Barycentric Interpolation Mapping
# Type 'y' after 'do:' to perform barycentric interpolation mapping.
  do: {y}
-------------------------------------------------------------------------
2.3 RBF Interpolation Mapping
```

```
# Type 'y' after 'do:' to perform RBF interpolation mapping.
# Input the desired RBF function after 'type of RBF function:', choices are:
# - multiquadric,
# - thinplate,
# - gauss,
# - inversemultiquadric,
# - compact.
  do: {y}
  type of RBF function: {compact}
  r0: {0.4}
------------------------------------------------------------------------
2.4 Kriging Mapping
# Type 'y' after 'do:' to perform a kriging mapping.
  do: {y}
------------------------------------------------------------------------
2.5 Nurbs Surface Interpolation Mapping
# Type 'y' after 'do:' to perform a nurbs surface interpolation mapping.
  do: {y}
  patch size: {5}
  patch degree: {2}
```

If the mode "performance" is chosen, then the file `PerformanceTest.txt` will be read by the program to do a performance test. The file `PerformanceTest.txt` locates in the same directory as the executable program. An example of file is listed below. This file is very similar to the file `setting.txt`. The difference is that the program can automatically run the mapping multiple times from different meshes to a single mesh. The purpose is to enable a single launch to test the error and the computation time with increasing refinement, which is used in Chapter 4. The data to be mapped will be generated by the program automatically (e.g. $f(x, y, z) = \sin(x) + \sin(y) + \sin(z)$). The program will not write any `.res` file that contains the mapping result, instead, it will prompt the error and computation time for each mapping method. The class PerformanceTest is in charge of parsing the file, but it will not be explained here.

```
# The user input place is between '{' and '}'.
========================================================================
1. Input data
# Please input the mesh files with known results after 'meshes from:',
#   and the mesh file without result after 'mesh to:'.
  meshes from: {sphere10x10.msh} {sphere20x20.msh} {sphere40x40.msh}
  mesh to: {sphere1000x1000.msh}
========================================================================
2. Setting mapping methods

2.1 Nearest Neighbor Mapping
# Type 'y' after 'do:' to perform nearest neighbor mapping.
  do: {y}
------------------------------------------------------------------------
```

```
2.2 Barycentric Interpolation Mapping
# Type 'y' after 'do:' to perform barycentric interpolation mapping.
  do: {y}
-------------------------------------------------------------------------
2.3 RBF Interpolation Mapping
# Type 'y' after 'do:' to perform RBF interpolation mapping.
# Input the desired RBF function after 'type of RBF function:', choices are:
# - multiquadric,
# - thinplate,
# - gauss,
# - inversemultiquadric,
# - compact.
  do: {y}
  type of RBF function: {compact}
  r0: {0.4}
-------------------------------------------------------------------------
2.4 Kriging Mapping
# Type 'y' after 'do:' to perform a kriging mapping.
  do: {y}
-------------------------------------------------------------------------
2.5 Nurbs Surface Interpolation Mapping
# Type 'y' after 'do:' to perform a nurbs surface interpolation mapping.
  do: {y}
  patch size: {5}
  patch degree: {2}
```

What is prompted by running the program with the above file is listed below.

```
Choose running mode: mapping or performance (m/p)?
p

Nearest neighbor mapping:
           time       err(max)        node id
           0.47        1.35549         275519
           0.62       0.650982         247751
            0.8       0.313808         272370

Barycentric interpolation mapping:
           time       err(max)        node id
           1.92       0.515195         307042
           2.53       0.237144         234127
           2.83       0.115606         299173

RBF interpolation mapping:
           time       err(max)        node id
          17.08      0.0665389         912945
          68.51      0.0128652         913255
```

```
         277.71     0.00242224         914568

Kriging mapping:
           time       err(max)        node id
          16.69       0.077032         454902
          65.33      0.0246947           5964
         264.27      0.0105193           7493

Nurbs surface interpolation mapping:
           time       err(max)        node id
          46.09      0.0801293         263514
          44.15      0.0168142         311027
          42.72      0.0046317          26864

The end!
```

4 Performance

The most important performance criteria of mapping are the error and the computation time. A mapping method is ideal if it can give small error within a short time period. Another criterion of mapping is the smoothness of the result, since zigzags will cause numerical problem in simulation of the physical model. This chapter will focus on the error convergence behavior of all mapping methods on different surfaces, and present the computation time and the smoothness.

When analyzing the error and the computation time, the mapping is performed only between four nodes' quadrilateral meshes, because only this kind of mesh is allowed in NURBS surface interpolation mapping. At the end of this chapter, the mapping between triangular meshes with all methods except NURBS surface interpolation mapping will be tested.

When showing the error and computation time in diagrams, the name of each mapping method is denoted by an abbreviation in the legend. As a result, "NN" represents nearest neighbor mapping, "BI" represents barycentric interpolation mapping, "NURBS" represents NURBS surface interpolation mapping, "RBF" represents RBF interpolation mapping and "Kriging" represents Kriging mapping.

There is one remark on the setting of some mapping methods. There are two parameters for setting NURBS surface interpolation mapping. The first one is the patch size n, which means the patch contains $n \times n$ nodes. The second one is the order of B-spline function p. During the test, two groups of parameters are used to test the effect of improving the patch size and the order of the B-spline function. These two groups are $n = 5, p = 2$ and $n = 7, p = 3$. There are also two parameters for setting RBF interpolation mapping, the type of RBF and the parameter $r0$ (see Section 2.4.1). Because there is no detailed instruction about how to choose the type of RBF and the value of $r0$, a generally "good" one is chosen, with *compactly supported* RBF and $r0 = 0.4$. There is only one variogram function implemented which is called *power*, so only Kriging mapping with *power* variogram is used in the test.

4.1 Storage and Singularity

Before presenting the error and the computation time, the storage problem for Kriging mapping and RBF interpolation mapping and the singularity problem for the latter must be discussed.

We use $\mathbf{A}$ to denote the mesh with given data, and $\mathbf{B}$ to denote the mesh whose data are going to be computed by mapping ($\mathbf{A}$ and $\mathbf{B}$ are obtained by meshing the same surface with different refinement). If $\mathbf{A}$ has 160×160 nodes, then the computer

must allocate $(160 \times 160)^2$ floating point numbers for the matrix in eq.(2.22) or eq.(2.33) when doing RBF interpolation mapping or Kriging mapping. The program uses double precision format for a floating point number (specified by "double" in the C language), which occupies 8 bytes memory [10]. So the memory required by the matrix is $(160 \times 160)^2 \times 8 \div 1024 \div 1024 \div 1024 = 4.88\mathbf{GB}$. And when $\mathbf{A}$ is refined by 320×320 nodes, the memory required is $(320 \times 320)^2 \times 8 \div 1024 \div 1024 \div 1024 = 78.13\mathbf{GB}$. But the total memory of the computer that runs the test cases is $7.8\mathbf{GB}$, so RBF interpolation mapping and Kriging mapping cannot give a result when $\mathbf{A}$ has equal or more than 320×320 nodes.

In addition to the storage problem, RBF interpolation mapping also has a singularity problem. Because in RBF interpolation mapping, the more nodes $\mathbf{A}$ has, the easier the matrix becomes singular, so that the linear equation system cannot be solved by LU-decomposition.

Because of these problems, Kriging mapping gives result with $\mathbf{A}$ having up to 160×160 nodes, while RBF interpolation mapping gives result with $\mathbf{A}$ having even less nodes due to the singularity. This is the reason why in the diagrams of error and computation time in the following sections, there are less sampling points in the curve for these two methods than other mapping methods.

4.2 Error

Before computing the error of mapping, the first task is to define a proper error. Assume the data field on $\mathbf{A}$ can be assigned by a function $f(x, y, z)$, so after mapping, the data field on $\mathbf{B}$ is naturally expected to be of the same function $f(x, y, z)$. And the error of mapping on a node (x_i, y_i, z_i) of $\mathbf{B}$ is defined by the absolute value of the difference between its data value and the expected value $f(x_i, y_i, z_i)$. To conveniently compare the error of mapping with different methods, the *maximum* of the errors on all nodes is used as a measurement of the error of a mapping method.

The error of mapping is dependent on several issues. First is the shape of the surface where the mapping is performed. There are five surfaces modeled in GiD to be used in testing the error, which are the sphere, the catenoid, the wave, the cylinder, and the plane. These surfaces are shown from Figure 4.1 to Figure 4.5, with the explanation about how they are created in GiD. The plane is the simplest surface with no curvature, and the cylinder is a "bended" plane with curvature only in one direction, while the other three surfaces are more complex with curvatures in both directions.

The error of mapping is also dependent on the refinement of $\mathbf{A}$ and $\mathbf{B}$. The more nodes $\mathbf{A}$ contains, the more discrete data values will be used in mapping, then smaller error is expected. While the more nodes $\mathbf{B}$ contains, the more samples of the data field on $\mathbf{B}$ are made, the chance of getting the "true" maximal error is bigger. As a result, $\mathbf{B}$ is generated by meshing different surfaces with 1000×1000 nodes, in order to provide somehow sufficient sampling points. And $\mathbf{A}$ is generated by meshing different surfaces with 10×10, 20×20, 40×40, 80×80, 160×160, 320×320, 640×640, 1280×1280 and 2560×2560 nodes, so that the convergence rate of the error associated with the

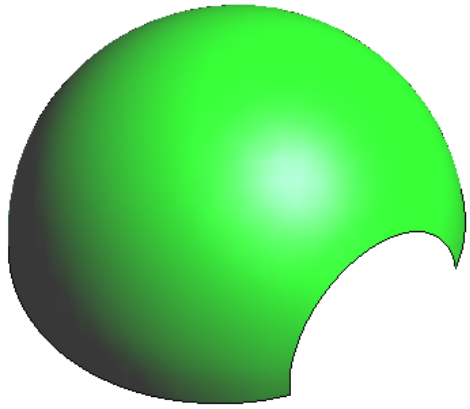


Figure 4.1: The sphere. It is a parametric surface created by setting $x = \sin(u)$, $y = \cos(u)\sin(v)$, $z = \cos(u)\cos(v)$ with $u \in [-\pi/3, \pi/3]$, $v \in [-\pi/2, \pi/2]$.

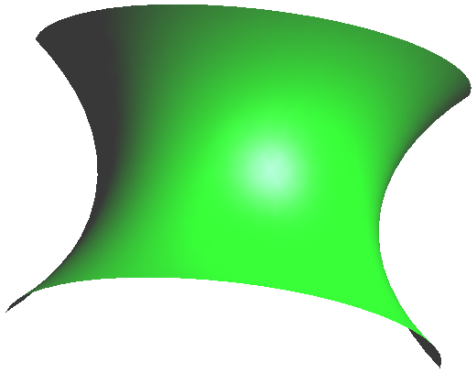


Figure 4.2: The catenoid. It is a parametric surface created by setting $x = 2\cos(u)\cosh(v/2)$, $y = v$, $z = 2\sin(u)\cosh(v/2)$ with $u \in [0, \pi]$, $v \in [-2, 2]$.

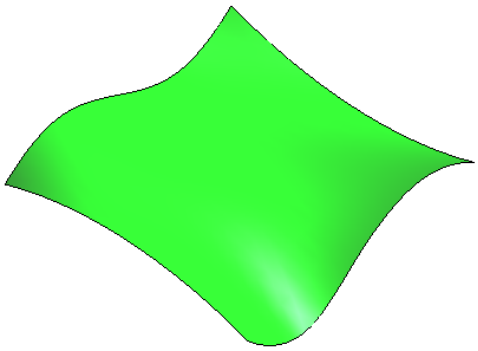


Figure 4.3: The wave. It is a NURBS surface created by four contour lines. The first line is a NURBS line that passes the points $(0, 0, 0)$, $(0.25, 0.1, 0)$, $(0.5, 0, 0)$, $(0.75, -0.1, 0)$, $(1, 0, 0)$. The second line is a NURBS line that passes the points $(0, 0, 1)$, $(0.25, -0.15, 1)$, $(0.5, 0, 1)$, $(0.75, 0.1, 1)$, $(1, 0, 1)$. The third line is an arc that passes the points $(0, 0, 0)$, $(0, 0.1, 0.5)$, $(0, 0, 1)$. The last line is an arc that passes the points $(1, 0, 0)$, $(1, -0.1, 0.5)$, $(1, 0, 1)$.

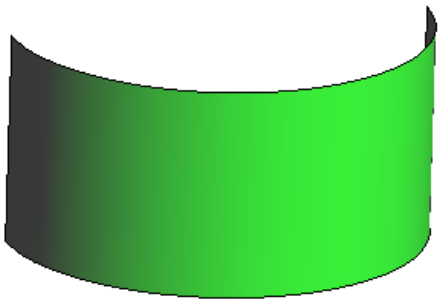


Figure 4.4: The cylinder. It is a parametric surface created by setting $x = \sin(u)$, $y = \cos(u)$, $z = v$ with $u \in [0, \pi]$, $v \in [0, 1]$.

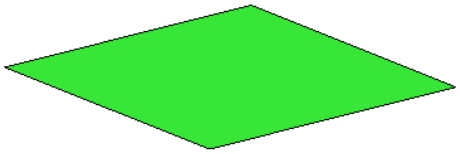


Figure 4.5: The plane. It is a plane constructed by four points $(0,0,0)$, $(1,0,0)$, $(1,1,0)$, $(0,1,0)$.

refinement of **A** can be shown in diagrams.

Another issue that affects the error is the data field that are mapped. In the following, three types of data fields are designed and used in testing the error, which are the constant data field, the linear data field, and the sine data field.

4.2.1 Constant Data Field

If the data field to be mapped is a constant data field, which means the data values on all nodes of **A** are equal to the same constant, then a good mapping method should give no mapping error. This can be theoretically guaranteed if a mapping method calculates the data value on a node of B by a linear combination of the data values on nodes of **A**, with the coefficients summed up to 1. This can be mathematically expressed as

$$y_{Bi} = \sum_{j=1}^{N_A} w_j y_{Aj} \qquad \text{with} \quad \sum_{j=1}^{N_A} w_j = 1, \tag{4.1}$$

where N_A is the number of nodes in **A**, and y_{Aj} represents the data value on a node of **A** while y_{Bi} represents the data value on a node of **B**, and w_j is the weight (coefficient). So if $y_{Aj} = c$ for all $j = 1, \ldots, N_A$ where c is a constant, then we have

$$y_{Bi} = \sum_{j=1}^{N_A} w_j y_{Aj} = c \sum_{j=1}^{N_A} w_j = c.$$

Nearest neighbor mapping fulfills eq.(4.1) because only one data value is used with the weight equal to one. Barycentric interpolation mapping and Kriging mapping also fulfills eq.(4.1) as shown respectively in eq.(2.2) and eq.(2.28). NURBS surface interpolation mapping fulfills eq.(4.1) too, but more explanation is needed. One property of the NURBS curve in eq.(2.5) is [6]

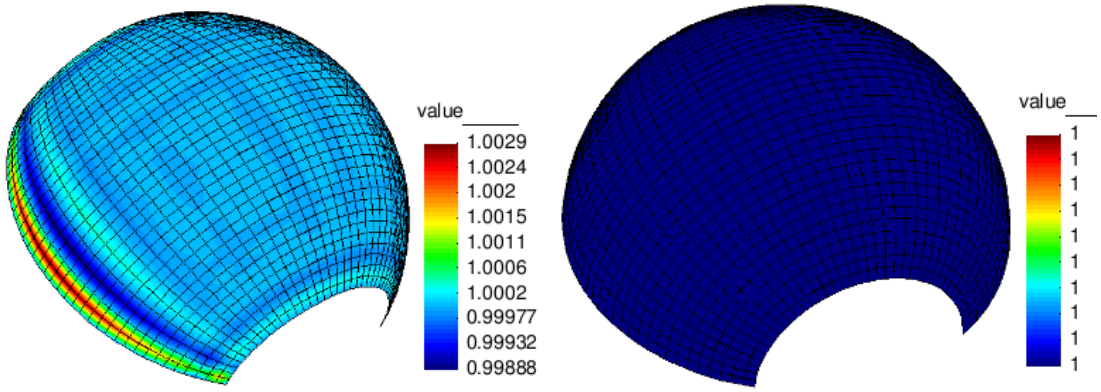
$$\sum_{i=0}^{n} N_{i,p}(u) = 1 \qquad 0 \le u \le 1, \tag{4.2}$$

which will not be proved here. B-spline surface in eq.(2.8) has the similar property because

$$\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) = \sum_{i=0}^{n} N_{i,p}(u) \sum_{j=0}^{m} N_{j,q}(v) = \sum_{i=0}^{n} N_{i,p}(u) = 1. \tag{4.3}$$

With eq.(4.3) we know that, if in eq.(2.8) we substitute control point $\mathbf{P}_{i,j}$ by a constant c (assume c is the data value on this control point), then the left hand side is equal to c constantly. This means if the control points of a NURBS surface have the same data value c, then the data value on an arbitrary point of this surface is equal to c. The data values on all control points are computed by interpolating the points in a NURBS surface patch, as shown in eq.(2.14). Again with eq.(4.3) we know that, if in eq.(2.14) we substitute $\mathbf{Q}_{k,l}$ by a constant c, then $\mathbf{P}_{i,j} = c$ for all $i = 0, \ldots, n$ and $j = 0, \ldots, m$ is the solution for eq.(2.14). This means if all points in a NURBS surface patch have the same data value c, then all control points of the NURBS surface interpolating these points will have the same data value c. So now it is proved that constant data field can be exactly mapped by NURBS surface interpolation mapping due to the property shown in eq.(4.3). However, RBF interpolation mapping does not have the similar property, because it computes the data value by the linear combination of radial basis functions instead of the data values as shown in eq.(2.20).

The mapping of constant data field is tested by running the program *MeshMapping* . So $\mathbf{A}$ is generated by meshing the sphere with 20×20 nodes and $\mathbf{B}$ is generated by meshing the sphere with 40×40 nodes. The data field $f(x, y, z) = 1$ is assigned on $\mathbf{A}$ and to be mapped to $\mathbf{B}$. All mapping methods except RBF interpolation mapping can map the data field exactly as shown in Figure 4.6.



(a) Result of RBF interpolation mapping (b) Result of other mapping methods

Figure 4.6: The results of mapping constant data field $f(x, y, z) = 1$ on the sphere using different mapping methods.

4.2.2 Linear Data Field

Now the data field on mesh **A** is set by a simple linear function $f(x, y, z) = x + y + z$ for different surfaces as shown respectively in Figure 4.7, Figure 4.9, Figure 4.11, Figure 4.13 and Figure 4.15.

On each surface, different mapping methods are performed to map the linear data field from **A** with increasing refinement. The errors of different methods are drawn in the same diagram for each surface, as shown respectively in Figure 4.8, Figure 4.10, Figure 4.12, Figure 4.14 and Figure 4.16. From these figures, we can observe that

- The error of all mapping methods become smaller when **A** has finer mesh, which means the errors of all mapping methods are convergent to zero.
- The error curves of NURBS surface interpolation mapping with different patch sizes (n in the legend) and different orders of B-spline functions (p in the legend) are nearly overlapped, this means improving the patch size and the order of B-spline function will not necessarily yield smaller error.
- The errors of NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping are the smallest in general, and the error of nearest neighbor mapping is the biggest, while the error of barycentric interpolation mapping is generally in between. But when mapping on the plane, the error of barycentric interpolation mapping is close to zero.
- The error curves of nearest neighbor mapping and NURBS surface interpolation mapping are both approximately straight lines with the same slope, which indicates the convergence rate of nearest neighbor mapping and NURBS surface interpolation mapping are the same. And because when we refine **A** from $n \times n$ nodes to $2n \times 2n$ nodes, the error is approximately halved, then the error of these two mapping methods is proportional to $O(1/n)$ (n is the square root of the number of nodes in **A**). When mapping on the cylinder the error of barycentric interpolation mapping is approximately proportional to $O(1/n^2)$ because when we refine **A** from $n \times n$ nodes to $2n \times 2n$ nodes the error will generally become a quarter of the original case. The error curve of barycentric interpolation mapping is not linear for surfaces with curvature in both directions. For RBF interpolation mapping and Kriging mapping, it is not possible to judge the convergence rate since there are few sampling points.

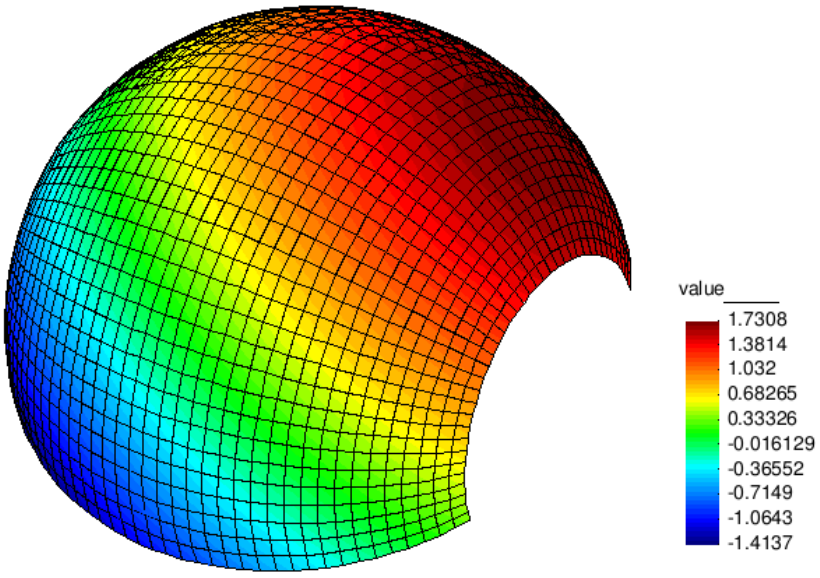


Figure 4.7: The linear data field on the sphere.

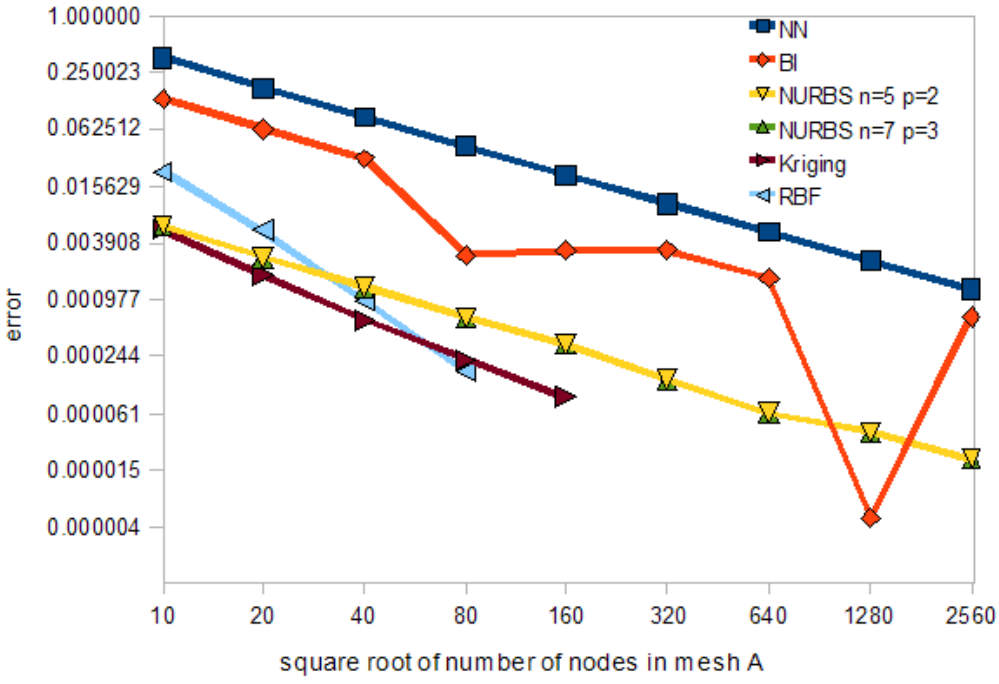


Figure 4.8: The error of mapping linear data field on the sphere using different mapping methods.

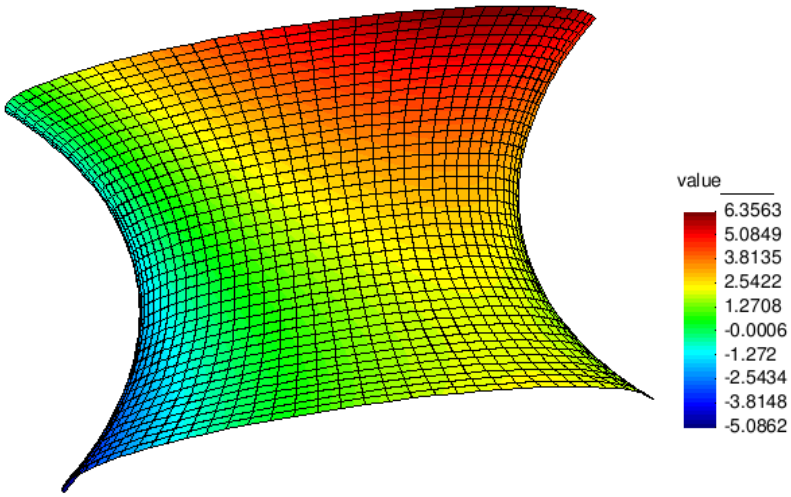


Figure 4.9: The linear data field on the catenoid.

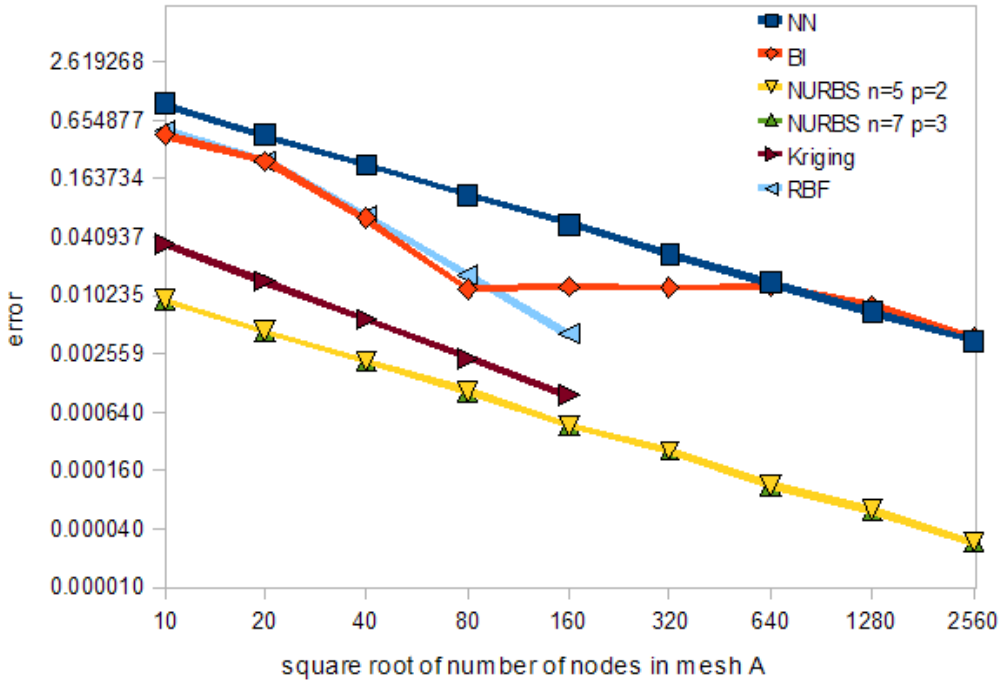


Figure 4.10: The error of mapping linear data field on the catenoid using different mapping methods.

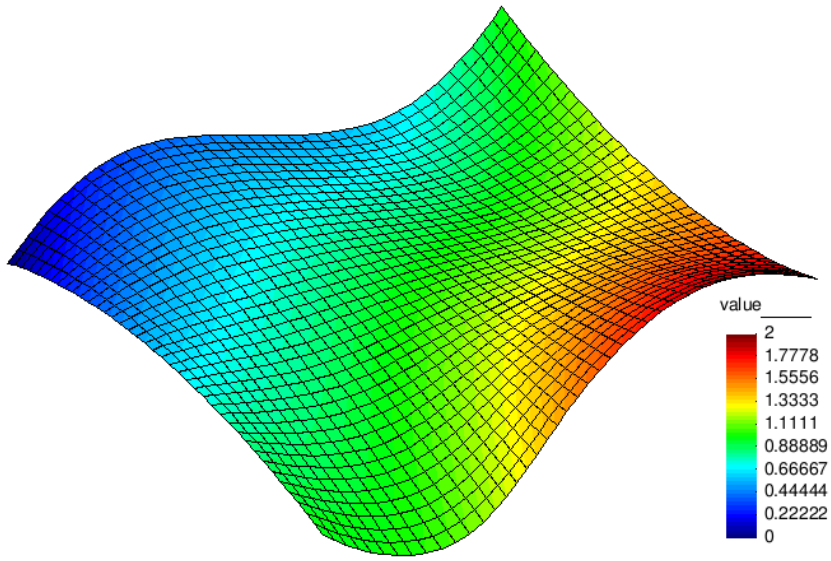


Figure 4.11: The linear data field on the wave.

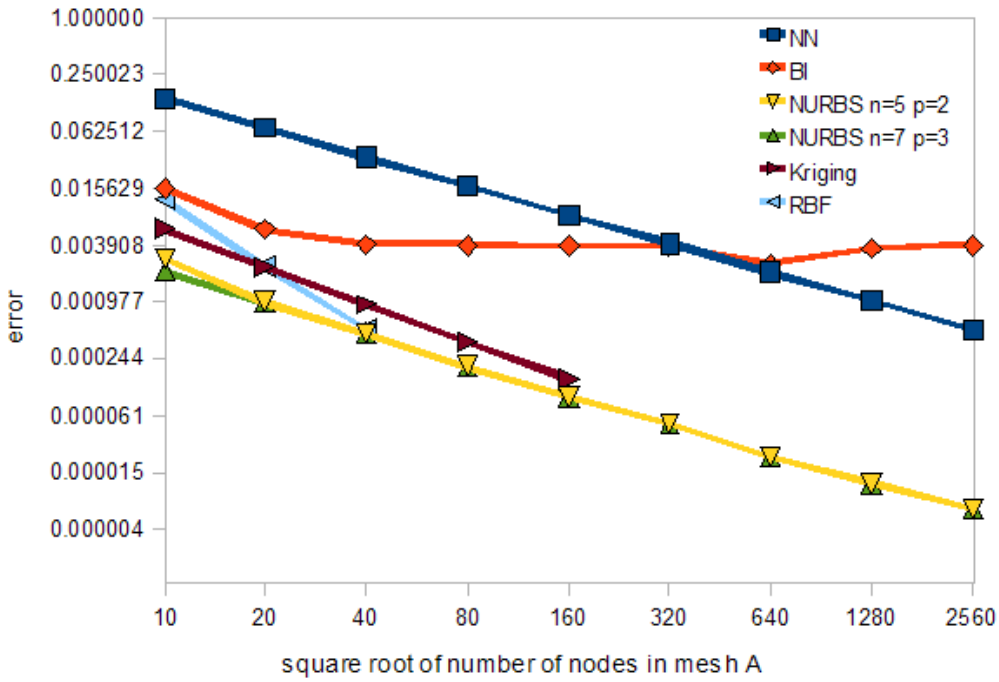


Figure 4.12: The error of mapping linear data field on the wave using different mapping methods.

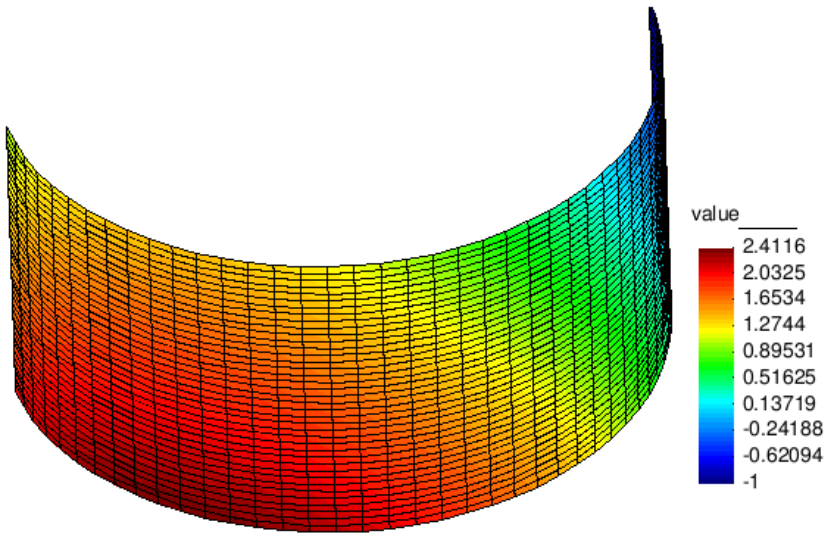


Figure 4.13: The linear data field on the cylinder.

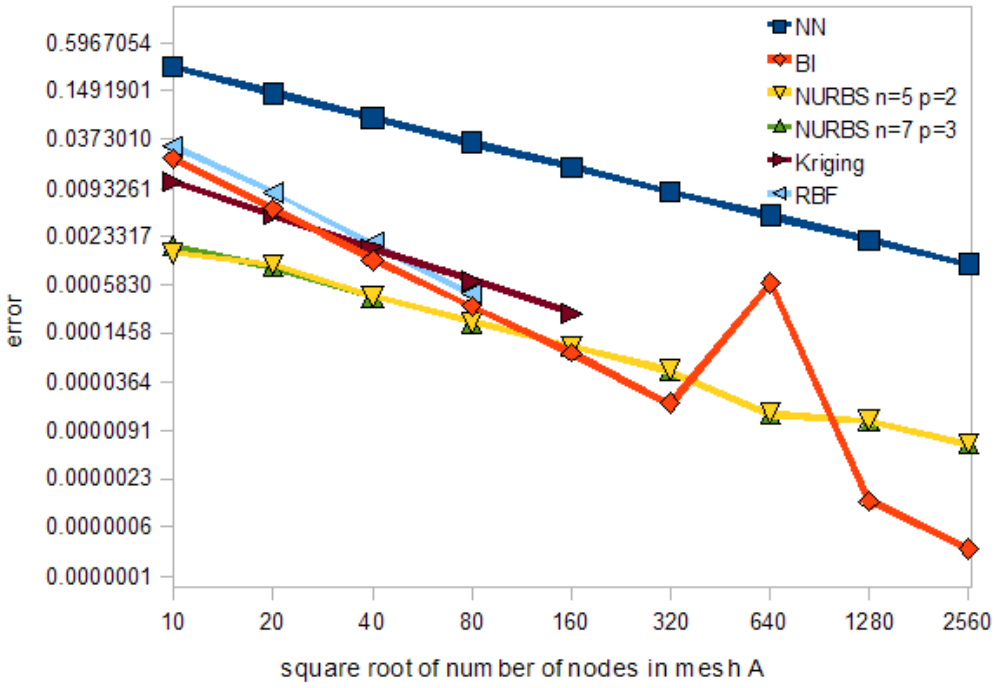


Figure 4.14: The error of mapping linear data field on the cylinder using different mapping methods.

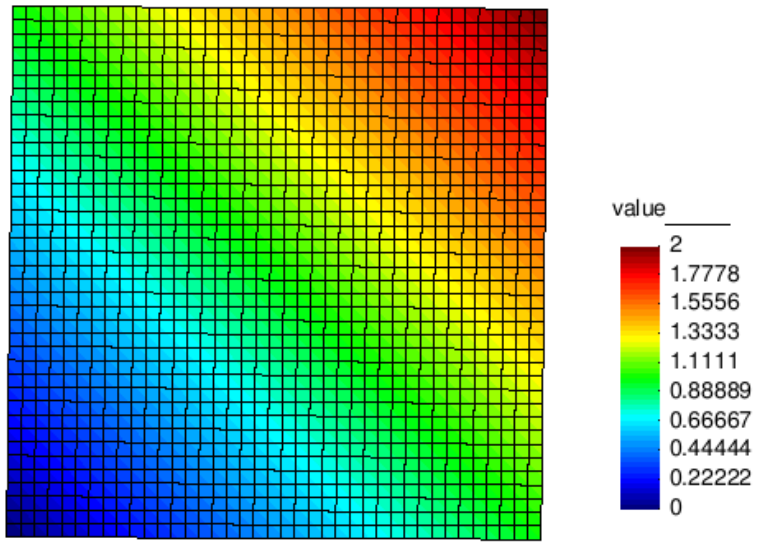


Figure 4.15: The linear data field on the plane.

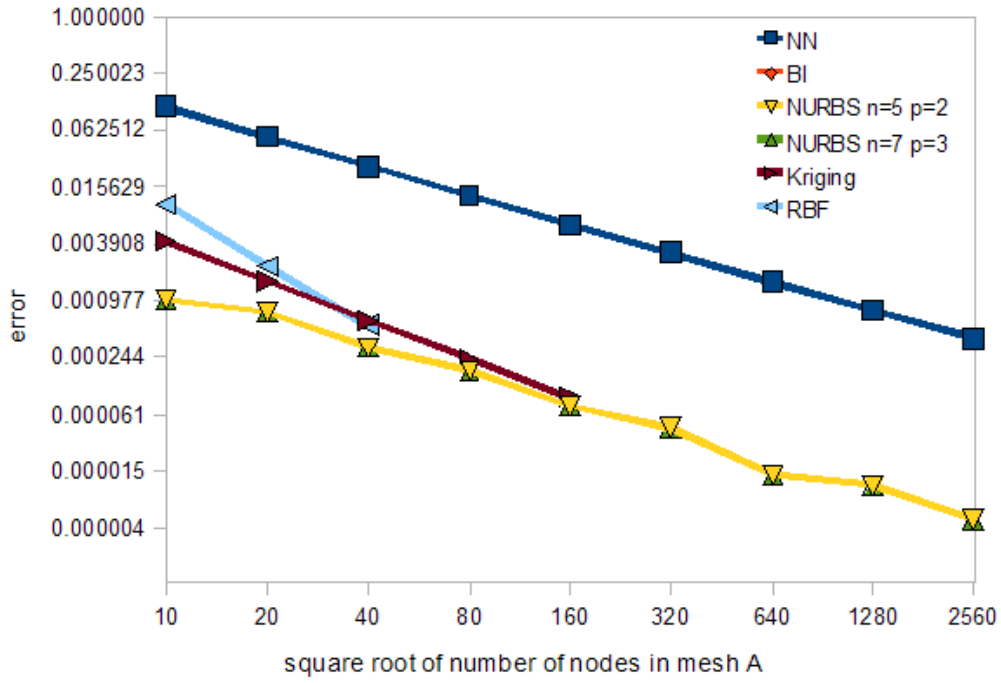


Figure 4.16: The error of mapping linear data field on the plane using different mapping methods. The error curve of barycentric interpolation mapping is missing in the diagram because it gives errors smaller than 10^{-13}, which cannot be shown with the logarithmic scale of the error. But compared with the errors of other mapping methods, the errors of barycentric interpolation can be regarded as zero.

4.2.3 Sine Data Field

Now the data field on **A** is set by a sine function $f(x, y, z) = a\sin(x) + a\sin(y) + a\sin(z)$ with a a scalar which has different value for different surfaces as shown respectively in Figure 4.17, Figure 4.19, Figure 4.21, Figure 4.23 and Figure 4.25. The idea of setting the value of a is to make both peak and valley of the data field occur at least once on the surface.

On each surface, different mapping methods are performed to map the sine data field from A with increasing refinement to B. Then the errors of different methods are drawn in the same diagram for each surface, as shown respectively in Figure 4.18, Figure 4.20, Figure 4.22, Figure 4.24 and Figure 4.26. From these figures, we can observe that

- The errors of all mapping methods become smaller when mesh **A** has finer mesh, which means the error of all mapping methods are convergent to zero.
- The error curves of NURBS surface interpolation mapping with different patch sizes (n in the legend) and different orders of B-spline functions (p in the legend) are nearly overlapped, this means improving the patch size and the order of B-spline function will not necessarily yield smaller error.
- The errors of NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping are the smallest, and the error of nearest neighbor mapping is the biggest, while the error of barycentric interpolation mapping is generally in between.
- The error curves of nearest neighbor mapping and NURBS surface interpolation mapping are both approximately straight lines with the same slope. The error of these two mapping methods is proportional to $O(1/n)$ (n is the square root of the number of nodes in **A**). When mapping on the cylinder and the plane the error of barycentric interpolation mapping is approximately proportional to $O(1/n^2)$. But the error curve of barycentric interpolation mapping is not linear for surfaces with curvature in both directions. For RBF interpolation mapping and Kriging mapping, it is not possible to judge the convergence rate since there are few sampling points.

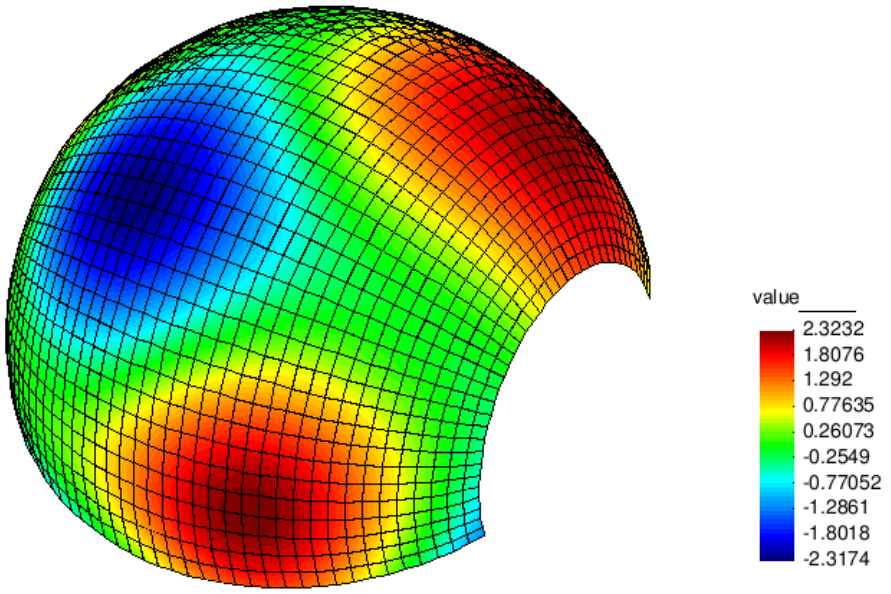


Figure 4.17: The sine data field on the sphere ($f(x, y, z) = 4\sin(x) + 4\sin(y) + 4\sin(z)$).

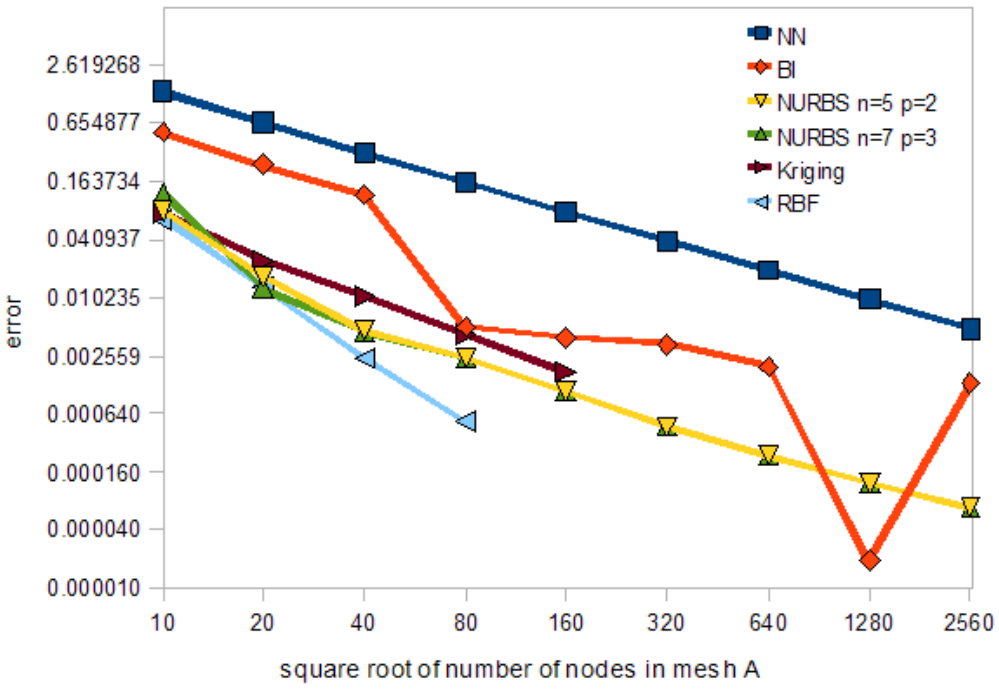


Figure 4.18: The error of mapping sine data field on the sphere using different mapping methods.

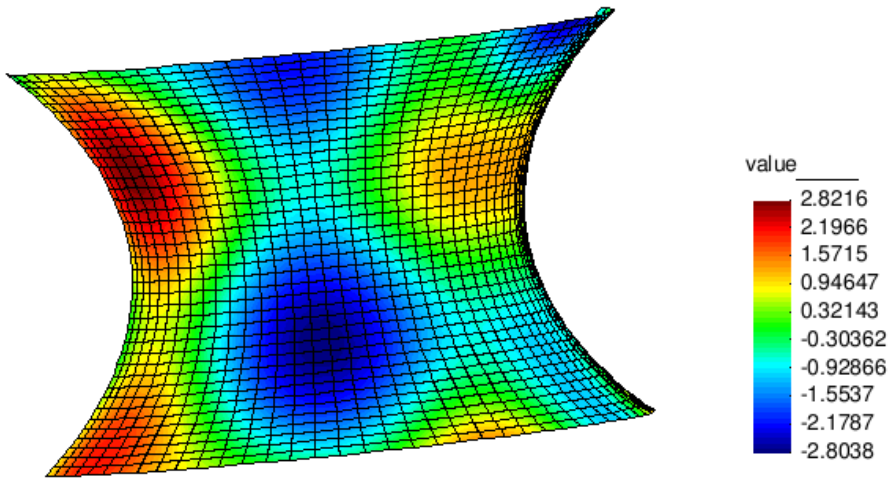


Figure 4.19: The sine data field on the catenoid ($f(x, y, z) = 2\sin(x)+2\sin(y)+2\sin(z)$).

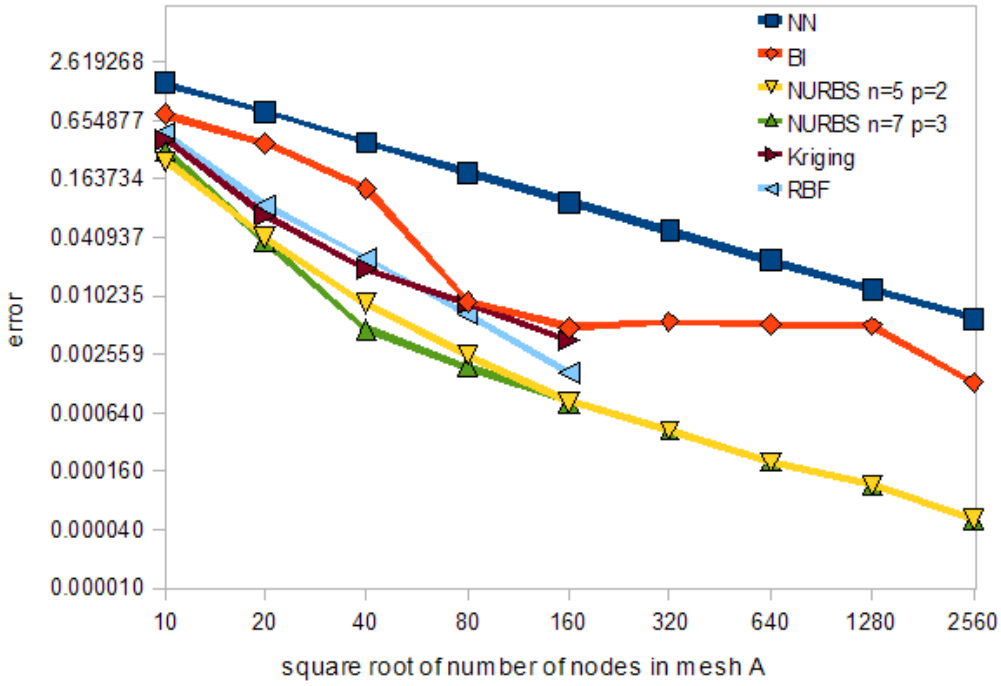


Figure 4.20: The error of mapping sine data field on the catenoid using different mapping methods.

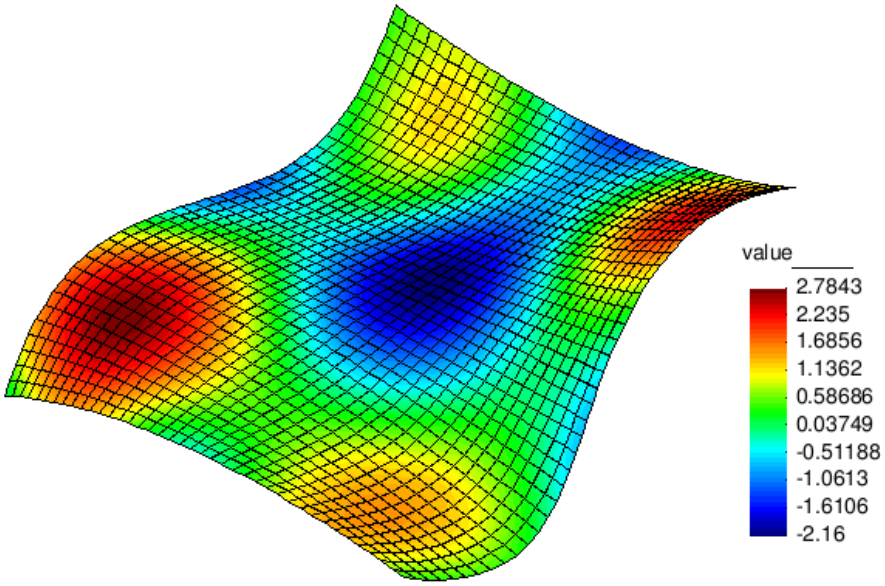


Figure 4.21: The sine data field on the wave ($f(x, y, z) = 9\sin(x) + 9\sin(y) + 9\sin(z)$).

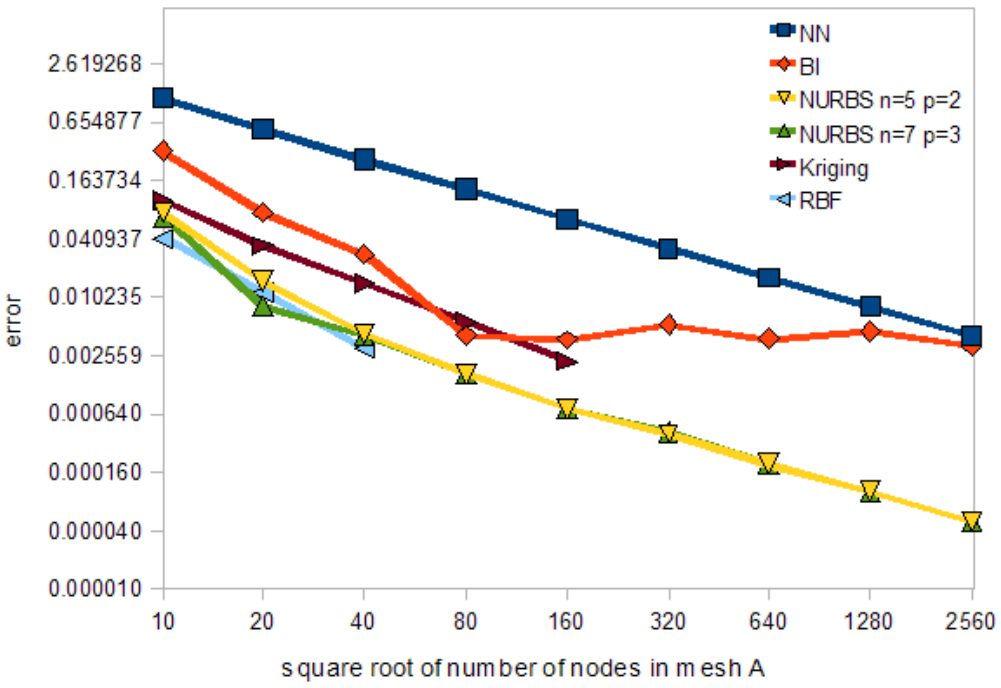


Figure 4.22: The error of mapping sine data field on the wave using different mapping methods.

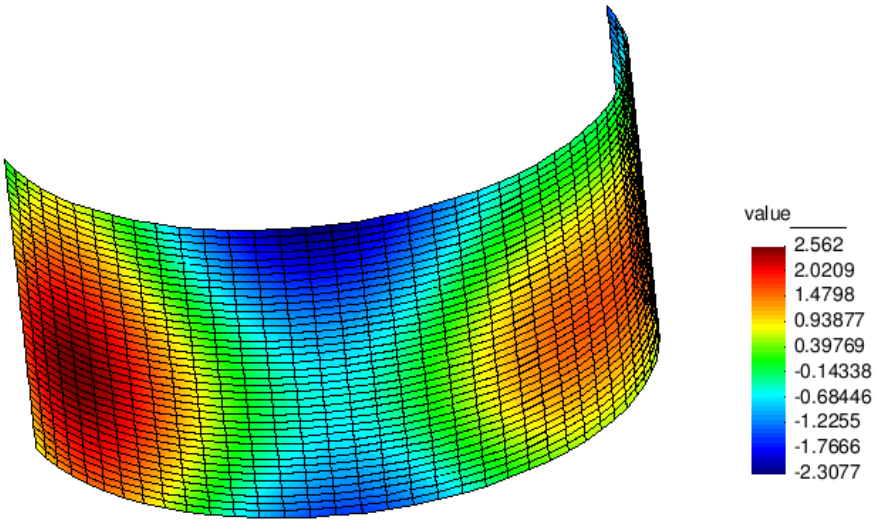


Figure 4.23: The sine data field on the cylinder ($f(x, y, z) = 4\sin(x)+4\sin(y)+4\sin(z)$).

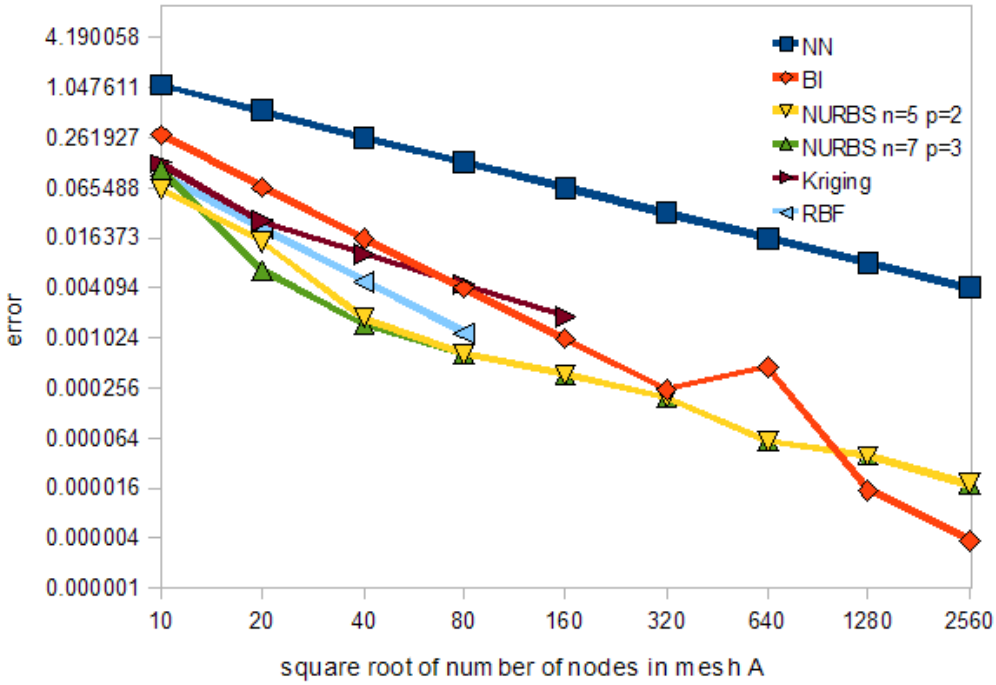


Figure 4.24: The error of mapping sine data field on the cylinder using different mapping methods.

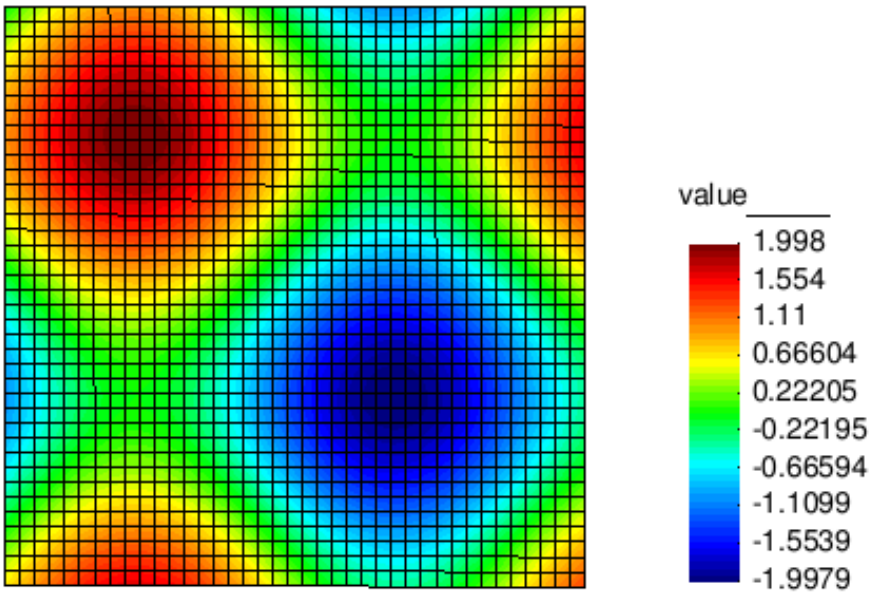


Figure 4.25: The sine data field on the plane ($f(x, y, z) = 7\sin(x) + 7\sin(y) + 7\sin(z)$).

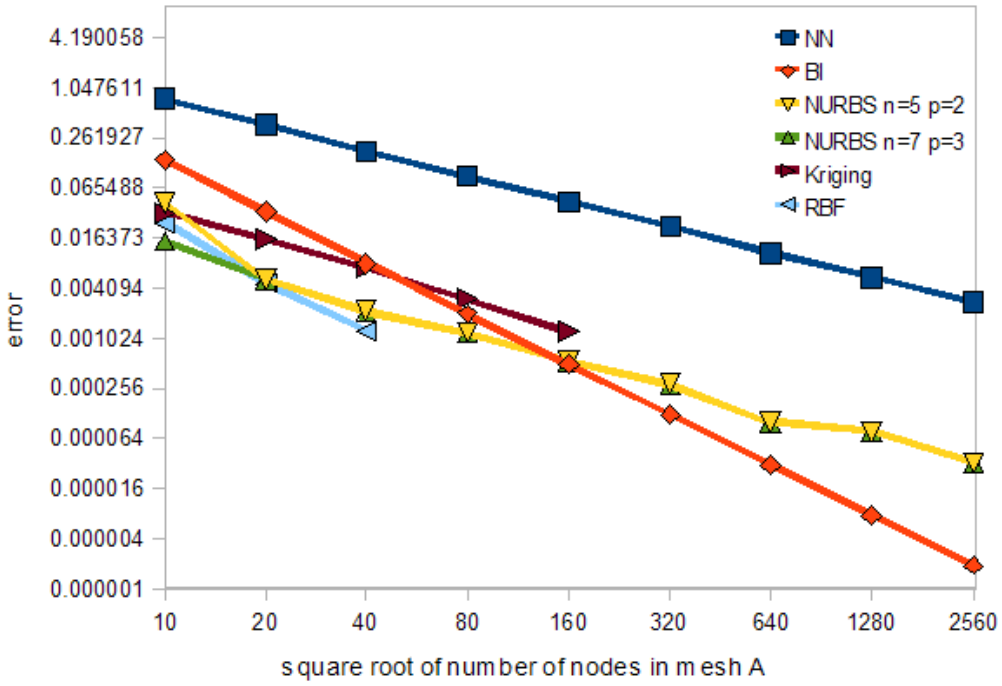


Figure 4.26: The error of mapping sine data field on the plane using different mapping methods.

4.2.4 Summary

In this section, the error behaviors of all mapping methods with different data fields are summarized below:

- All methods except RBF interpolation mapping can exactly map the constant data field.
- For mapping on non-constant data field, NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping give the most accurate result, and nearest neighbor mapping gives the least accurate result, while barycentric interpolation mapping does it generally in between. Within the limited test cases of RBF interpolation mapping and Kriging mapping, they do not give significantly more accurate result than NURBS surface interpolation mapping.
- Improve the patch size and the order of B-spline function in NURBS surface interpolation mapping will not necessarily yield smaller error.
- For mapping on non-constant data field, the error of nearest neighbor mapping and NURBS surface interpolation mapping is proportional to $O(1/n)$ (n is the square root of the number of nodes in $\mathbf{A}$). The shape of the error curve of barycentric interpolation mapping is sensitive to the shape of the surface.
- Barycentric interpolation mapping gives more accurate result on the cylinder and the plane.

4.3 Computation time

The computation time reflects the complexity of the algorithm of each mapping method, which is introduced in Chapter 2. It is independent of the data field and the shape of the surface, it is only dependent on the number of nodes in $\mathbf{A}$ and $\mathbf{B}$. So here the surface is fixed to be the sphere, and the data field is set to the same as that in Figure 4.17. For drawing a curve of the computation time, $\mathbf{A}$ is generated by meshing the sphere separately with 10×10, 20×20, 40×40, 80×80, 160×160, 320×320, 640×640, 1280×1280 and 2560×2560 nodes. Figure 4.27 shows the computation time when $\mathbf{B}$ is generated by meshing the sphere with 1000×1000 nodes. Figure 4.28 shows the computation time when $\mathbf{B}$ is generated by meshing the sphere with 100×100 nodes. The CPU of the computer running the tests is a Intel 2 Quad processor Q9505 with clock speed 2.83GHz. From the figures we can observe that

- Nearest neighbor mapping costs the smallest computation time while barycentric interpolation mapping slightly higher. NURBS surface interpolation mapping costs moderate computation time, which is larger than barycentric interpolation mapping. RBF interpolation mapping and Kriging mapping cost the largest computation time, and when we refine $\mathbf{A}$ from $n \times n$ nodes to $2n \times 2n$ nodes, the computation time goes up significantly.

- For NURBS surface interpolation mapping, larger patch size and higher order of B-spline function of the NURBS surface will result in longer computation time.
- For nearest neighbor mapping, barycentric interpolation mapping and NURBS surface interpolation mapping, when the number of nodes in **A** is much larger than in **B**, then the computation time of the three methods is more or less the same, because then the time is mainly spent on building the kd-tree.

4.4 Smoothness of Data

The smoothness of data is also an important criterion for evaluating the performance of mapping. For example, if the displacement field on a surface is mapped, then the smoothness of the displacement field obtained from mapping will determine the smoothness of the deformed surface. If there are many zigzags on the deformed surface, then it may cause severe numerical problem in the simulation of a certain physical field. The zigzags always happen when mapping from a coarser mesh to a finer mesh.

We start from an illustrative test case. A simple strip is meshed to generate a coarser mesh and a finer mesh, as shown in Figure 4.29. The displacement field on the coarser mesh comes from a sine function over the length direction, and this field is mapped to the finer mesh with different mapping methods. The strips after deformation are shown in Figure 4.30. What is interesting is the boundary of the deformed strip in the length direction. In the coarser mesh, the boundary is a broken line. For the finer mesh, nearest neighbor mapping gives a stepped line boundary, and barycentric interpolation mapping gives a broken line boundary that follows the boundary of the coarser mesh, while the other mapping methods give smooth curve boundary. This indicates that NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping have a smoothening effect on the data field.

Now it comes a more delicate test case. The sphere is meshed by 20×20 nodes and 40×40 nodes respectively to generate **A** and **B**. The data field is the same as those in Figure 4.17, the difference is that we make the field be the displacement in z direction, in order to visualize the shape of the sphere after deformation. The deformed sphere of **A** is shown in Figure 4.31 and the deformed sphere of **B** obtained by different mapping methods are shown from Figure 4.32 to Figure 4.36. From these figures we can see, nearest neighbor mapping has severe zigzag problem, and there are only slight zigzags within a certain region by barycentric interpolation mapping, and for the rest mapping methods, the smoothness "looks" good (it is not mathematically strict to judge the smoothness of a surface by the outlook).

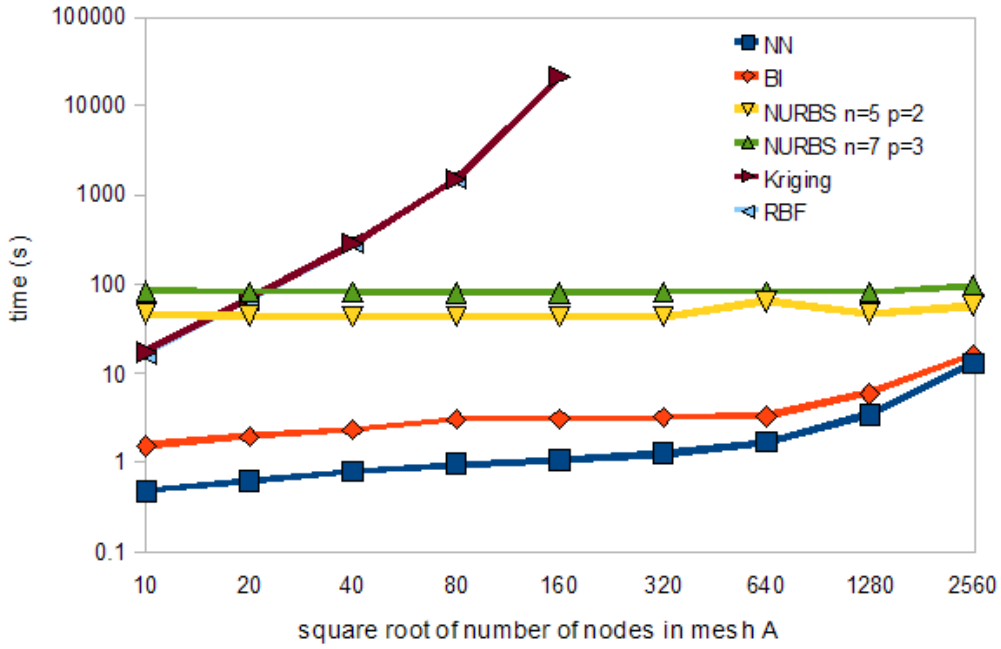


Figure 4.27: The computation time of different mapping methods when **B** is meshed by 1000×1000 nodes.

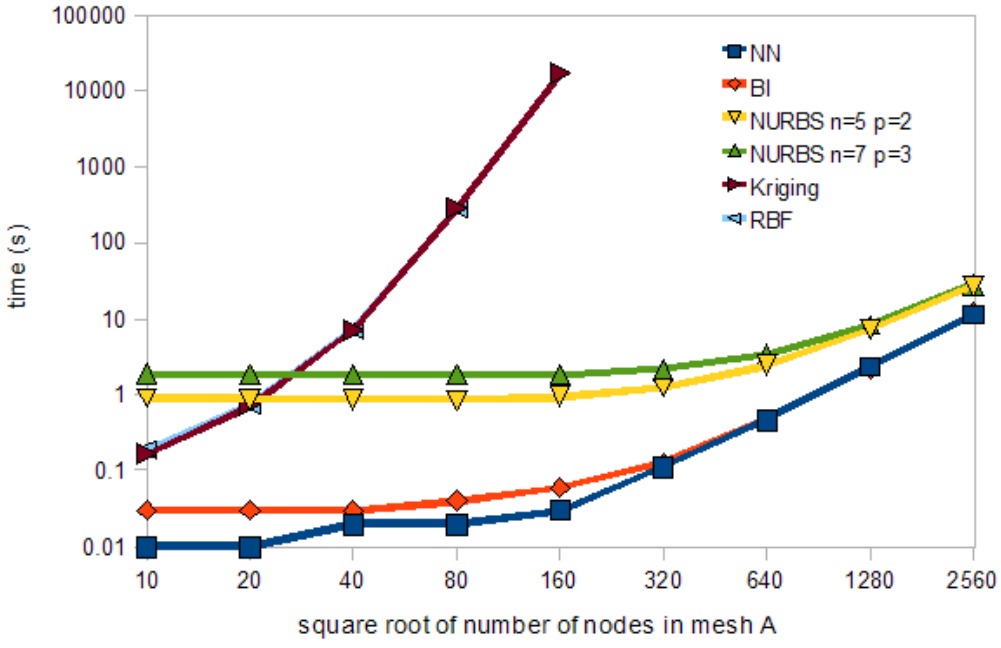
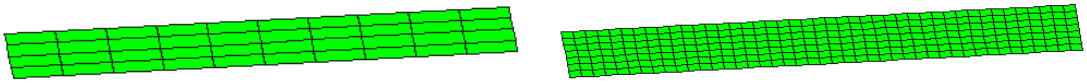
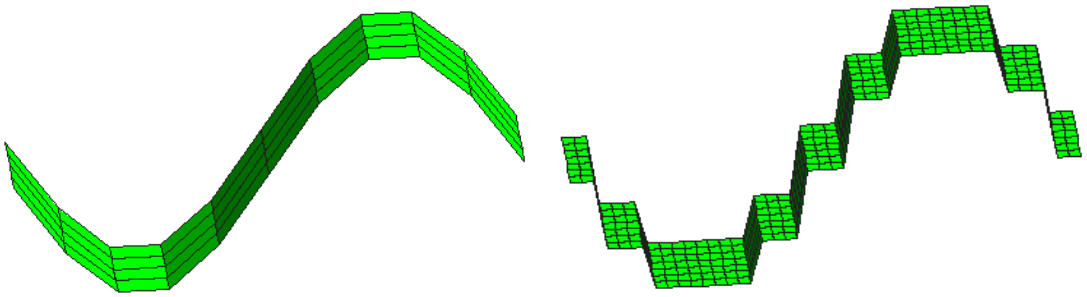


Figure 4.28: The computation time of different mapping methods when **B** is meshed by 100×100 nodes.

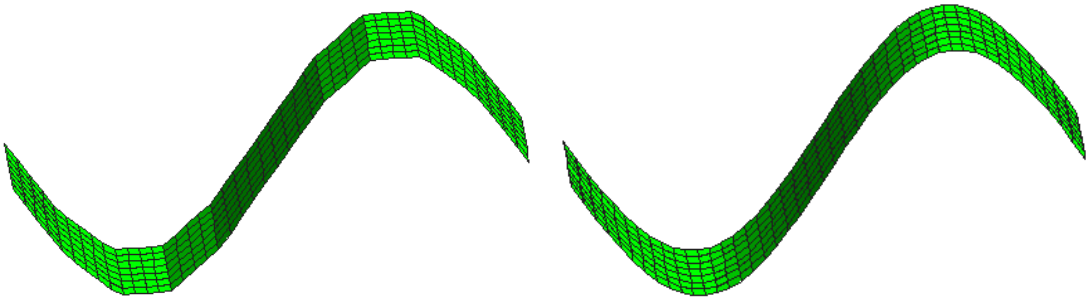


(a) Meshed by 3×10 elements (b) Meshed by 7×43 elements

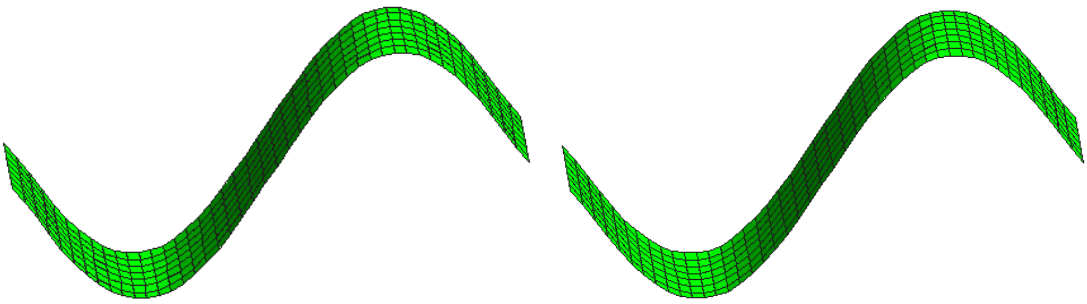
Figure 4.29: The coarser mesh and the finer mesh of the same strip.



(a) The coarser mesh after deformation (b) From nearest neighbor mapping



(c) From barycentric interpolation mapping (d) From NURBS surface interpolation mapping



(e) From RBF interpolation mapping (f) From Kriging mapping

Figure 4.30: The deformed strip of the coarser mesh, and the deformed strips of the finer mesh obtained from different mapping methods.

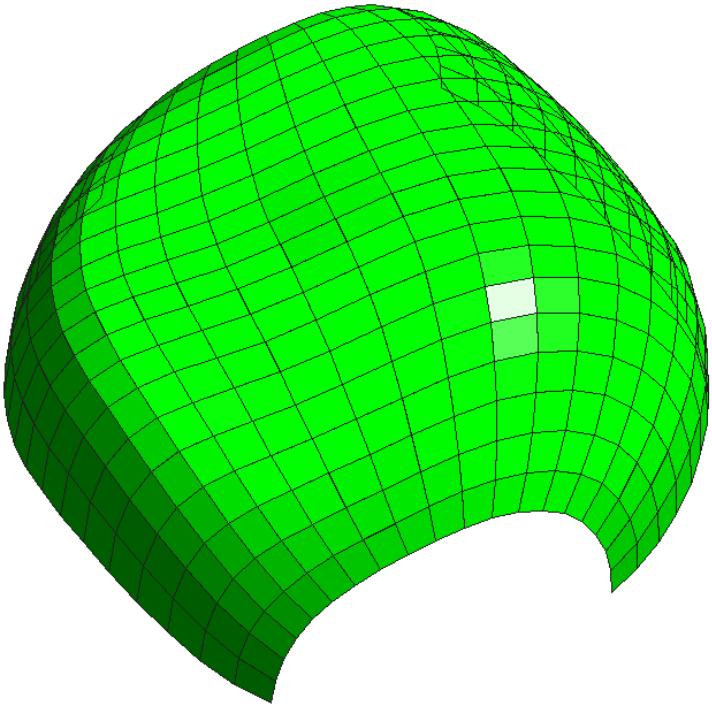


Figure 4.31: The deformed sphere of **A**. The displacements are calculated by $f(x, y, z) = 4\sin(x) + 4\sin(y) + 4\sin(z)$.

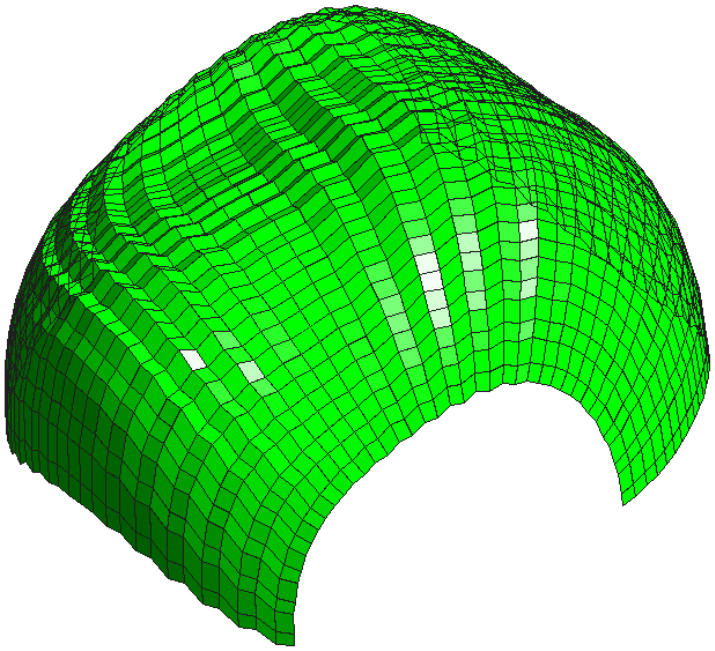


Figure 4.32: The deformed sphere of **B**. The displacements are obtained by nearest neighbor mapping.

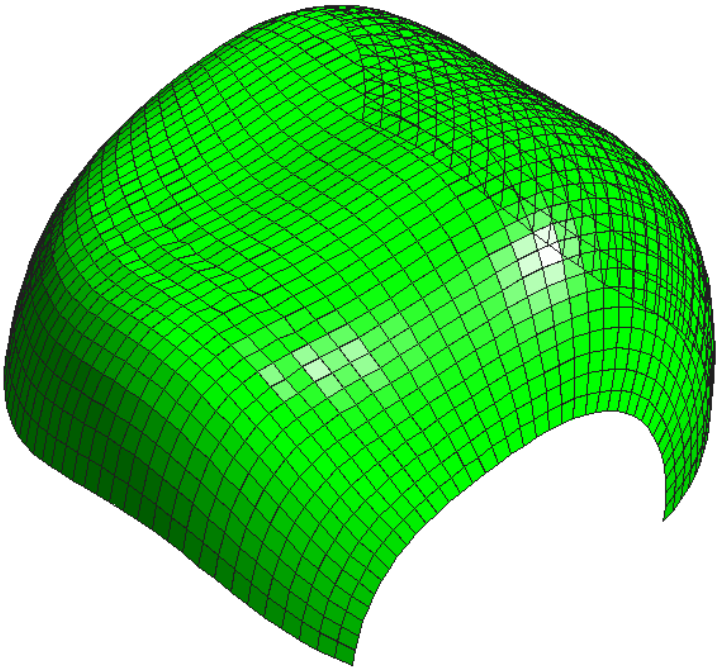


Figure 4.33: The deformed sphere of **B**. The displacements are obtained by barycentric interpolation mapping.

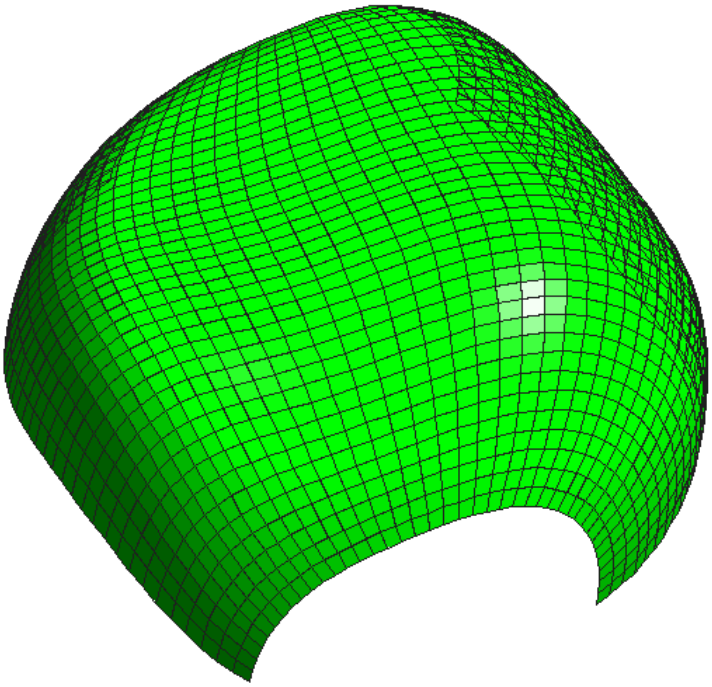


Figure 4.34: The deformed sphere of **B**. The displacements are obtained by NURBS surface interpolation mapping.

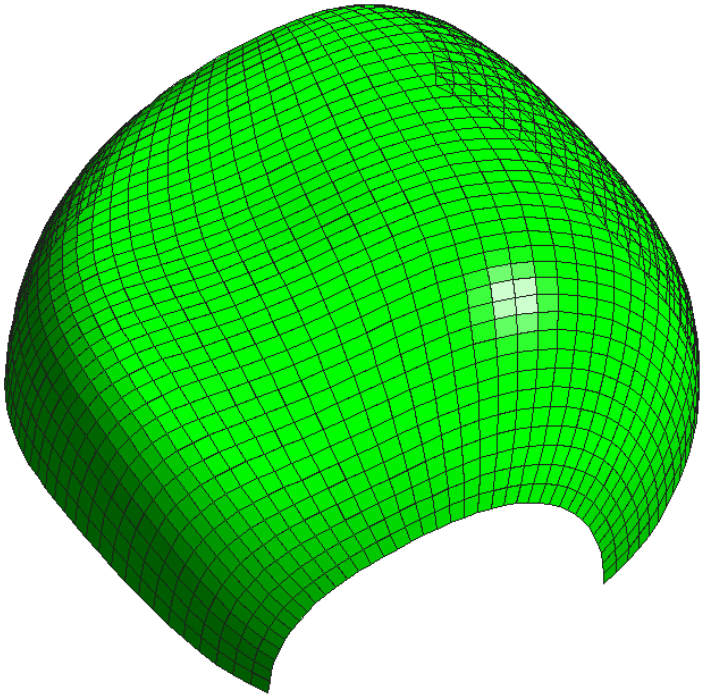


Figure 4.35: The deformed sphere of **B**. The displacements are obtained by RBF interpolation mapping.

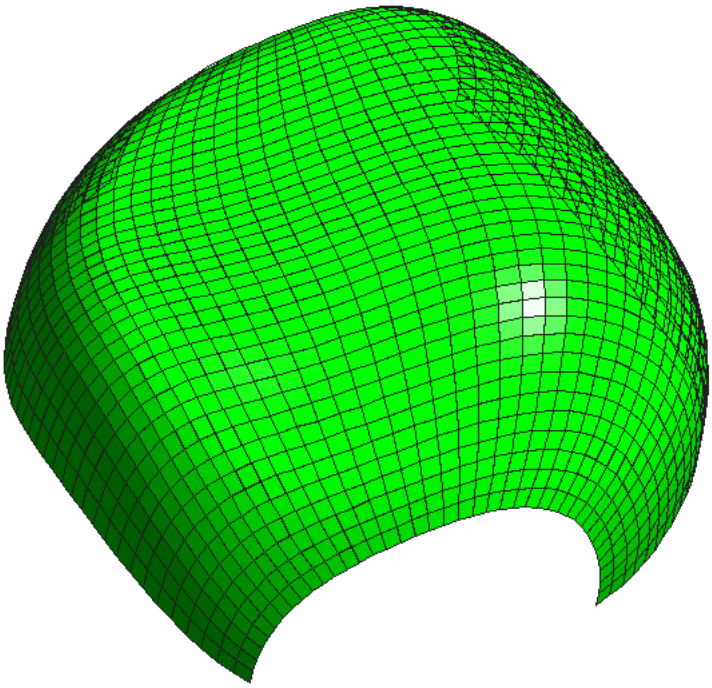


Figure 4.36: The deformed sphere of **B**. The displacements are obtained by Kriging mapping.

4.5 Mapping between Triangular Meshes

The surface used is a *Costa minimal surface*, which was discovered in 1982 by the Brazilian mathematician Celso Costa [11]. The surface is meshed by 4034 triangular elements and 1205 triangular elements respectively to generate **A** and **B** (see Figure 4.37 and Figure 4.38), which are from the work of Falko Dieringer. The data field $f(x, y, z) = \sin(0.6x) + \sin(0.6y) + \sin(z)$ is mapped, as shown in Figure 4.39. The errors of different mapping methods (except NURBS surface interpolation mapping) are listed in the table below, and the error fields are shown from Figure 4.40 to Figure 4.43 as well. As mapping between quadrilateral meshes, RBF interpolation mapping and Kriging mapping give smaller error than barycentric interpolation mapping, while nearest neighbor mapping gives the biggest error.

Mapping methods	Error
Nearest neighbor mapping	0.218
Barycentric interpolation mapping	0.098
RBF interpolation mapping	0.061
Kriging mapping	0.035

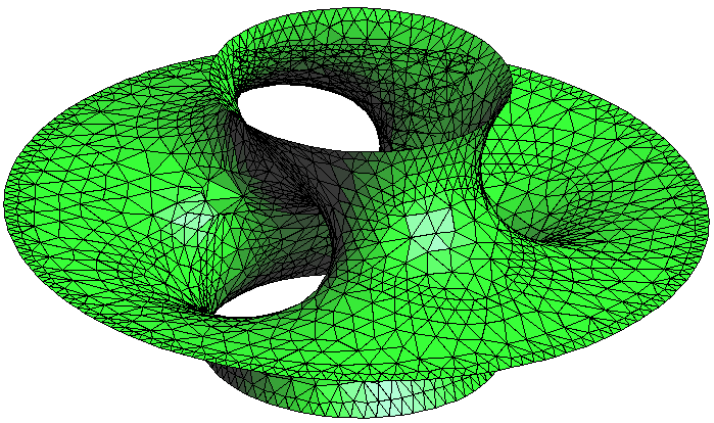


Figure 4.37: Mesh **A** which is generated by meshing the Costa surface with 4034 triangular elements.

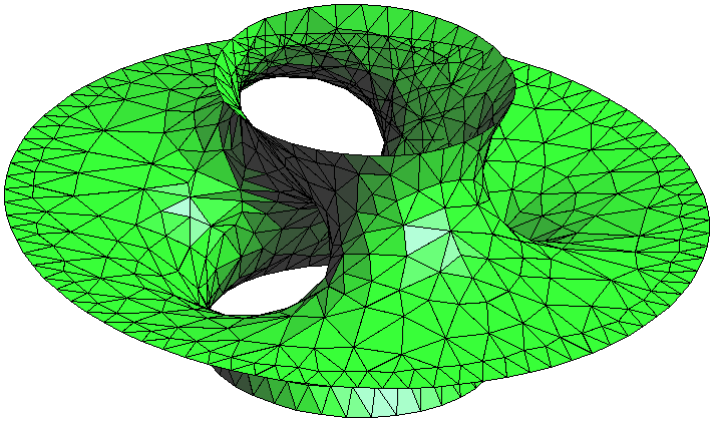


Figure 4.38: Mesh **B** which is generated by meshing the Costa surface with 1205 triangular elements.

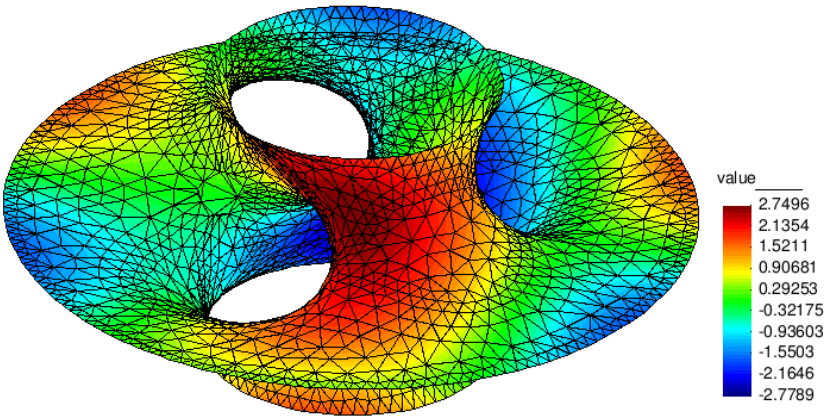


Figure 4.39: The data field $f(x, y, z) = \sin(0.6x) + \sin(0.6y) + \sin(z)$ on **A**.

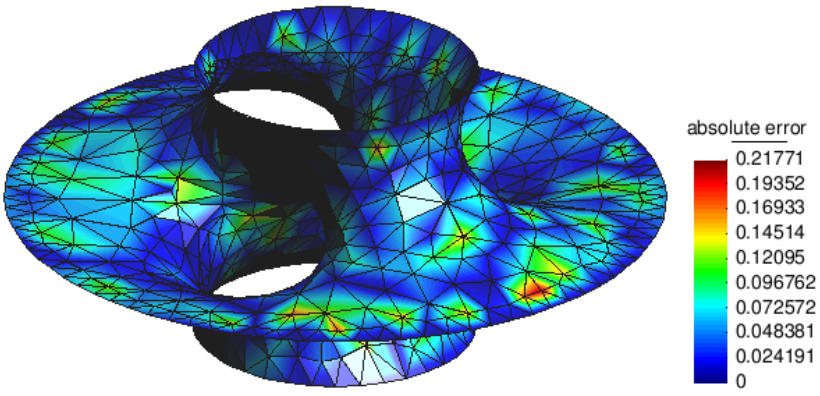


Figure 4.40: The error field from nearest neighbor mapping.

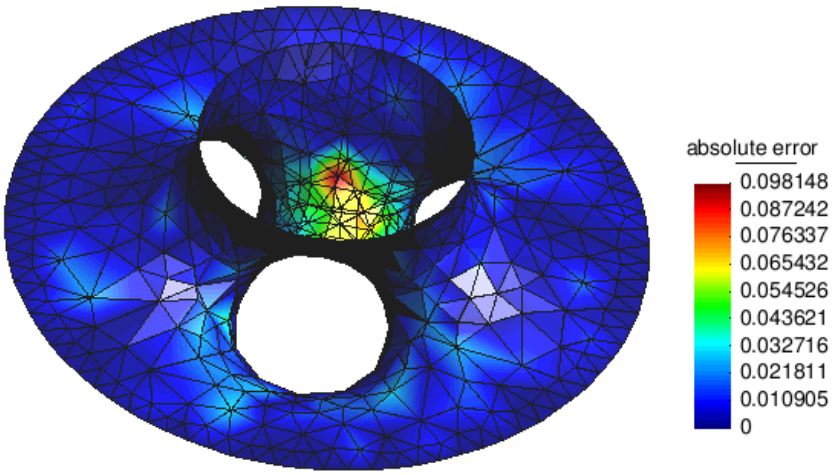


Figure 4.41: The error field from barycentric interpolation mapping.

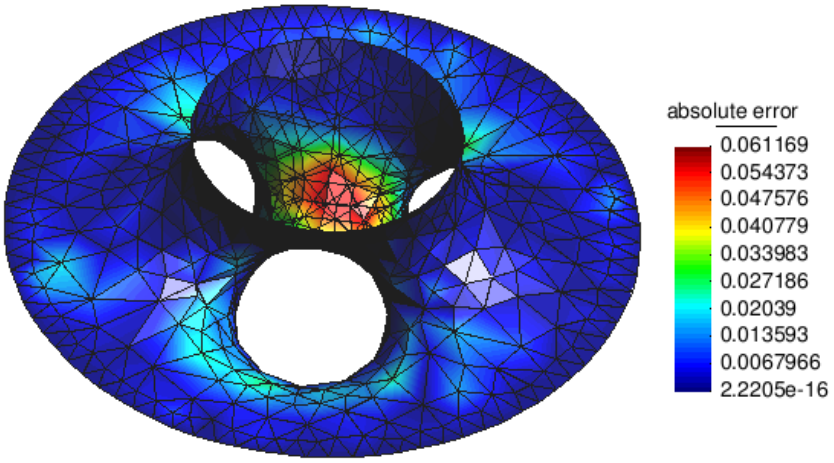


Figure 4.42: The error field from RBF interpolation mapping.

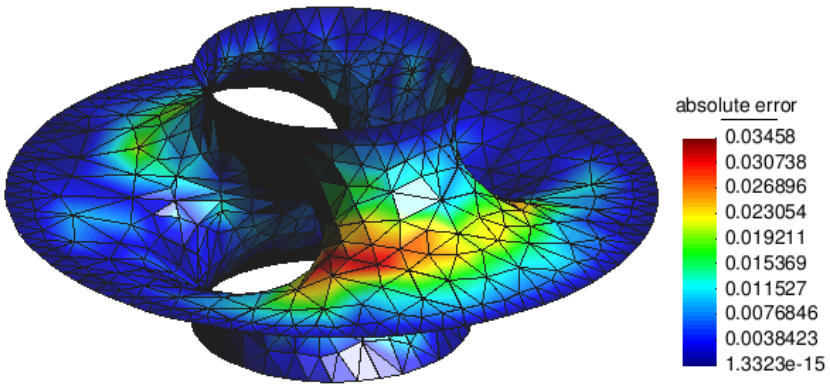


Figure 4.43: The error field from Kriging mapping.

5 Conclusion

This thesis is devoted to the study of different mapping methods that are used in transferring data between non-matching grids. What is achieved in this work will be summarized in Section 5.1 with evaluation of the results. Section 5.2 will list several ideas that have not been implemented in this thesis, but are important to improve the current study in the future.

5.1 Summary of the Work

This thesis begins with the introduction to the theory of five mapping methods, which are nearest neighbor mapping, barycentric interpolation mapping, NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping. The former three mapping methods have one common characteristic, that the data value on a node is computed by interpolating the data values on the neighboring node/nodes. Getting the neighboring node/nodes is a typical nearest neighbor problem. A data structure kd-tree is used to search the nearest neighbor because of its high efficiency. The complexity of these three methods are the same, which is $O(N \cdot \log N)$ (assume both meshes have the same number of points N). Nearest neighbor mapping directly assign the data value to be the same as that on the nearest neighbor. Barycentric interpolation mapping and NURBS surface interpolation mapping compute the data value by constructing a triangle/NURBS surface using more neighboring nodes with the hope of improving the accuracy. Both RBF interpolation mapping and Kriging mapping do global interpolation, i.e. all data points are used to build a function model of the data field, and given the position of an arbitrary point, the data value on it can be computed directly by the function model. For both methods, an $N \times N$ linear equation system has to be solved. This thesis uses LU-decomposition to solve the linear equation system. But this method has complexity of $O(N^3)$, which is too large compared with the complexity of other three methods.

The program *MeshMapping* implements all the mapping methods. Besides, the software GiD is used to model and mesh the surfaces, as well as visualize the data on a mesh. GiD provides its own file format for the mesh and the data. To perform mapping, the program is able to create the data structures of the mesh and the data by reading the mesh files and the data files. And the data computed by mapping can be written to a file of GiD format, so that the result of mapping can be visualized.

The program *MeshMapping* is used to run on a big number of test cases to test the performance of each mapping method. NURBS surface interpolation mapping, RBF interpolation mapping and Kriging mapping give the smallest error and best smoothness

of data in general. But with the same amount of error, NURBS surface interpolation mapping costs much less computation time and storage than the other two methods (the matrix in RBF interpolation mapping and Kriging mapping is made of $N \times N$ float point numbers, which is too large compared to $O(N)$ for other three methods). However, the disadvantage of NURBS surface interpolation mapping is that it restricts the mesh with data to be of element type *four nodes' quadrilateral*. Nearest neighbor mapping gives the biggest error while costs the smallest computation time, and it will produce severe zigzags on the data field when mapping from coarser mesh to finer mesh. Barycentric interpolation mapping has better error performance on the simple surface like the plane and the cylinder. It will produce slight zigzags on the data field when mapping from coarser mesh to finer mesh. The computation time of it is between nearest neighbor mapping and NURBS surface interpolation mapping. From these results, we can summarize some hints on using these mapping methods, as listed below:

- When mapping from a mesh of element type four nodes' quadrilaterals, NURBS surface interpolation is the best choice. Setting the patch size to be 5 and the order of B-spline functions to be 2 is enough for good accuracy, because increasing these parameters will not yield more accurate result but will cost much larger time.
- When mapping on a surface which has small curvature, barycentric interpolation mapping is a good choice.
- If nearest neighbor mapping and barycentric interpolation mapping are used, be aware of the possible zigzags of the data field.
- RBF interpolation mapping and Kriging mapping is not efficient when mapping large number of data points with respect to the computation time and storage.

5.2 Future Work

Limited by the time of a Master thesis, there are still many things to do to improve the study, as listed below:

- Enable using NURBS surface interpolation mapping on other types of quadrilaterals and even triangles.
- Implement iterative solvers to solve the linear equation system in RBF interpolation mapping and Kriging mapping to reduce the computation time.
- Investigate on how to set the parameters for each type of RBF function, to obtain non-singular matrix and more accurate result.
- Investigate and implement more variogram functions for Kriging mapping.
- Parallelize the code of all mapping methods to reduce the computation time.

Bibliography

[1] David M. Mount, Sunil Arya. ANN: A Library for Approximate Nearest Neighbor Searching. Website. http://www.cs.umd.edu/~mount/ANN/.

[2] David M. Mount. ANN Programming Manual, version 1.1. University of Maryland, 2010.

[3] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software, 3(3):209-226, 1977.

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery. Numerical Recipes - The Art of Scientific Computing, 3rd edition. Cambridge University Press, 2007.

[5] A.de Boer, A.H. van Zuijlen, H. Bijl. Review of coupling methods for non-matching meshes. *Computer methods in applied mechanics and engineering*, 196 (2007) 1515-1525.

[6] Les Piegl, Wayne Tiller. The NURBS Book, 1st edition. Springer, 1995.

[7] Hans Wackernagel. Multivariate Geostatistics, 3rd edition. Springer, 2003.

[8] Chaoyi Lang. Kriging Iterpolation. Website. http://www.nbb.cornell.edu/neurobio/land/oldstudentprojects/cs490-94to95/clang/kriging.html.

[9] GiD - The Personal Pre and Post Processor. Website. http://gid.cimne.upc.es/.

[10] Floating point. Website. http://en.wikipedia.org/wiki/Floating_point.

[11] Costa's minimal surface. Website. http://en.wikipedia.org/wiki/Costa's_minimal_surface.