Authors

SIMO Package
User Manual

CONTENTS

GLOSSARY

BdryIdcs Indices of boundary control points.

BdryVals Values of boundary control points.

CompDofs Degree of freedoms in components.

CtrlPts Control Points.

CtrlPts3D Control point coordinates in Cartesian space.

CtrlPts4D Control point coordinates in homogeneous space.

Curv Curve.

Dof Number of degree of freedom per control point.

Drchlt Dirichlet.

El Element connectivity matrix.

Idcs Indices.

Idx Index.

KntVect Knot Vector.

MeshBdry Mesh of the Boundary.

NCtrlPts Number of control points.

NDof Number of degree of freedoms for a patch.

NEl Number of elements.

NElDir Number of element in each direction.

NEN Number of local basis functions.

Neuman Neumann.

NNP Number of total control points for a patch.

NSD Number of dimensional space.

NURBSBdry NURBS of the Boundary.

Surf Surface.

uqKntVect Unique knot values.

Vals Values.

Volu Volume.

CHAPTER 1

FUNDAMENTALS OF THE ISOGEOMETRIC ANALYSIS

1.1 Univariate B-splines

Knot vector and B-spline functions, refinement, spline derivatives

B-spline curves

1.2 Multivariate tensor product B-splines

A B-spline patch (curve, surface, volume) is essentially defined by these corresponding formulas

$$\mathbf{C}(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi)\mathbf{P}_i. \tag{1.1}$$

$$\mathbf{S}(\xi,\eta) = \sum_{i=1}^{n}\sum_{j=1}^{m} N_{i,p}(\xi)M_{j,q}(\eta)\mathbf{P}_{i,j}, \tag{1.2}$$

$$\mathbf{S}(\xi,\eta,\zeta) = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{l} N_{i,p}(\xi)M_{j,q}(\eta)L_{k,l}(\zeta)\mathbf{P}_{i,j,k}, \tag{1.3}$$

where $N_{i,p}(\xi)$, $M_{j,q}(\eta)$, $L_{k,l}(\zeta)$ are univariate basis functions, $\mathbf{P}$ is the 4-vector of control point coordinates.

Knot vectors, B-spline functions

NURBS geometries, single patch and multi–patch domains

1.3 IsoGeometric Analysis

Basic Idea and Fundamentals

Isogeometric Analysis in Detail

B-splines and NURBS as Basis Functions

Mesh Structure in Isogeometric Analysis

Geometry Mapping

Refinement strategies

CHAPTER 2

MATHEMATICAL MODELS AND VARIATIONAL FORMULATION

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.

* * *

2.1 Mathematical Models

Heat conduction

Continuum Mechanics

Truss

Beam

Plate

Shell

Gradient Elasticity

Limit Analysis

2.2 Variational Formulation

CHAPTER 3

IMPLEMENTATION

This chapter is mainly devoted to the implementation of SIMO package within MATLAB.

* * *

3.1 Data Representation

Knot vector(s) and Control Point's Coordinates

The main information in isogeometric analysis is given by knot vector(s) and control points. In SIMO Package, these data structures are stored as following

- Knot vectors are stored in a cell array. So,
  - For curve: a B-spline (NURBS) curve is parametrized by one knot vector

```
KntVect = {[ 0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1]}
```

  - For surface: a B-spline (NURBS) surface is parametrized by two knot vectors

```
KntVect = {[ 0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1],...
 [0, 0, 1, 1]}
```

  - For solid: a B-spline (NURBS) surface is parametrized by three knot vectors

```
KntVect = {[ 0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1],...
 [0, 0, 1, 1],...
 [0, 0, 0, 0.5, 0.5, 1, 1, 1]}
```

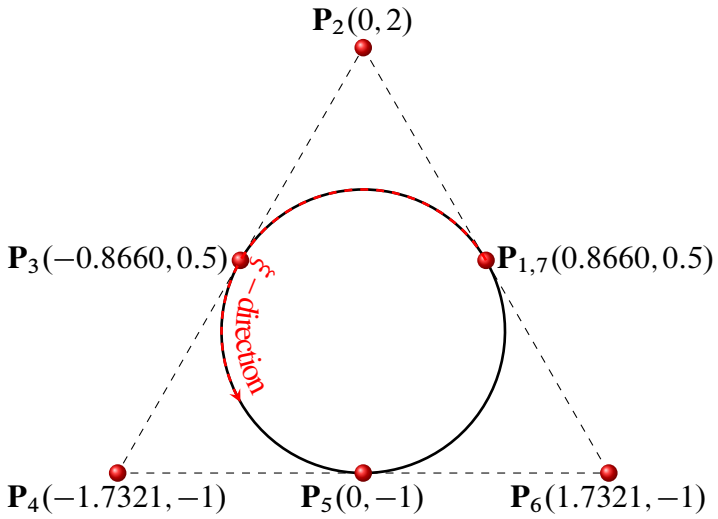
- Control points are stored in multi–dimensional array
  - For curve, coordinates (including weights) of control points are populated into a two–dimensional array, where each column contains coordinates of a control point (which together construct a control polyline). In SIMO Package, the size of the first dimension of this array should be always specified equal to 4 $(x \cdot w, y \cdot w, z \cdot w, w)$. This means the control point's data is added in form of homogeneous coordinates. The below code snippet is taken as an illustration for clarifying this point. Fig. 3.1 is a visualization of the way we store control point's coordinates.

```
radius = 1;

% Control point's coordinates in homogeneous space
```

```
CtrlPts = zeros(4, 7);
CtrlPts(1 : 2, 1, 1) = [cos(pi / 6); sin(pi / 6)];
CtrlPts(1 : 2, 2, 1) = [0; 1 / cos(pi / 3)];
CtrlPts(1 : 2, 3, 1) = [-cos(pi / 6); sin(pi / 6)];
CtrlPts(1 : 2, 4, 1) = [-tan(pi / 3); -1];
CtrlPts(1 : 2, 5, 1) = [0; -1];
CtrlPts(1 : 2, 6, 1) = [tan(pi / 3); -1];
CtrlPts(1 : 2, 7, 1) = [cos(pi / 6); sin(pi / 6)];

CtrlPts(1 : 3, :, :) = CtrlPts(1 : 3, :, :) * radius;
% assign the weights
CtrlPts(4, :, :) = 1;
W = 1 / 2;
CtrlPts(:, 2 : 2 : end, :) = CtrlPts(:, 2 : 2 : end, :) * W;
```



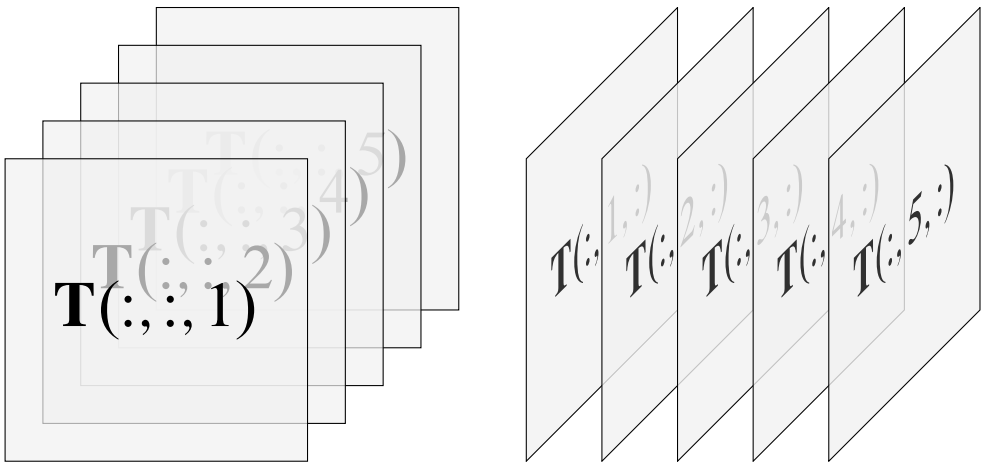
	$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$	$\mathbf{P}_4$	$\mathbf{P}_5$	$\mathbf{P}_6$	$\mathbf{P}_7$
x w	0.8660	0.0000	-0.8660	-0.8660	0.0000	0.8660	0.8660
y w	0.5000	1.0000	0.5000	-0.5000	-1.0000	-0.5000	0.5000
z w	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
w	1.0000	0.5000	1.0000	0.5000	1.0000	0.5000	1.0000

Figure 3.1: Demonstration of data storage for constructing a circle.

The arrow labelled ξ–direction in the Fig. 3.1 indicates the direction that parameter values ξ (knot values) grow from the lower to the higher ones (usually from 0 to 1 in practices), which correspond to the way from one end of the curve to the other (orient from the first to the second control point). This information is used to generate the curve and identify the direction on it (in order to determine the orientation of the curve – sign of jacobian of mapping, boundary's positions).

– For surface: Control point's coordinates are stored in three–dimensional array. Each frontal slice of the array represent a collection of control points which form the corresponding control polyline in the first direction and similarly for horizontal slices. Polylines in the first direction together with those in the second direction are linked to create the so-called control polygon of the considering surface. The images of these slices of the

three–dimensional array in MATLAB can be depicted as in the Fig. 3.2. A segment of code along with Fig. 3.3 are also given below to demonstrate how to declare this type of data structure in MATLAB and the corresponding visualization of this work.



(a) Frontal $\mathbf{T}(:,:,k)$ *(b) Lateral* $\mathbf{T}(:,j,:)$

Figure 3.2: Slices of a three–dimensional array

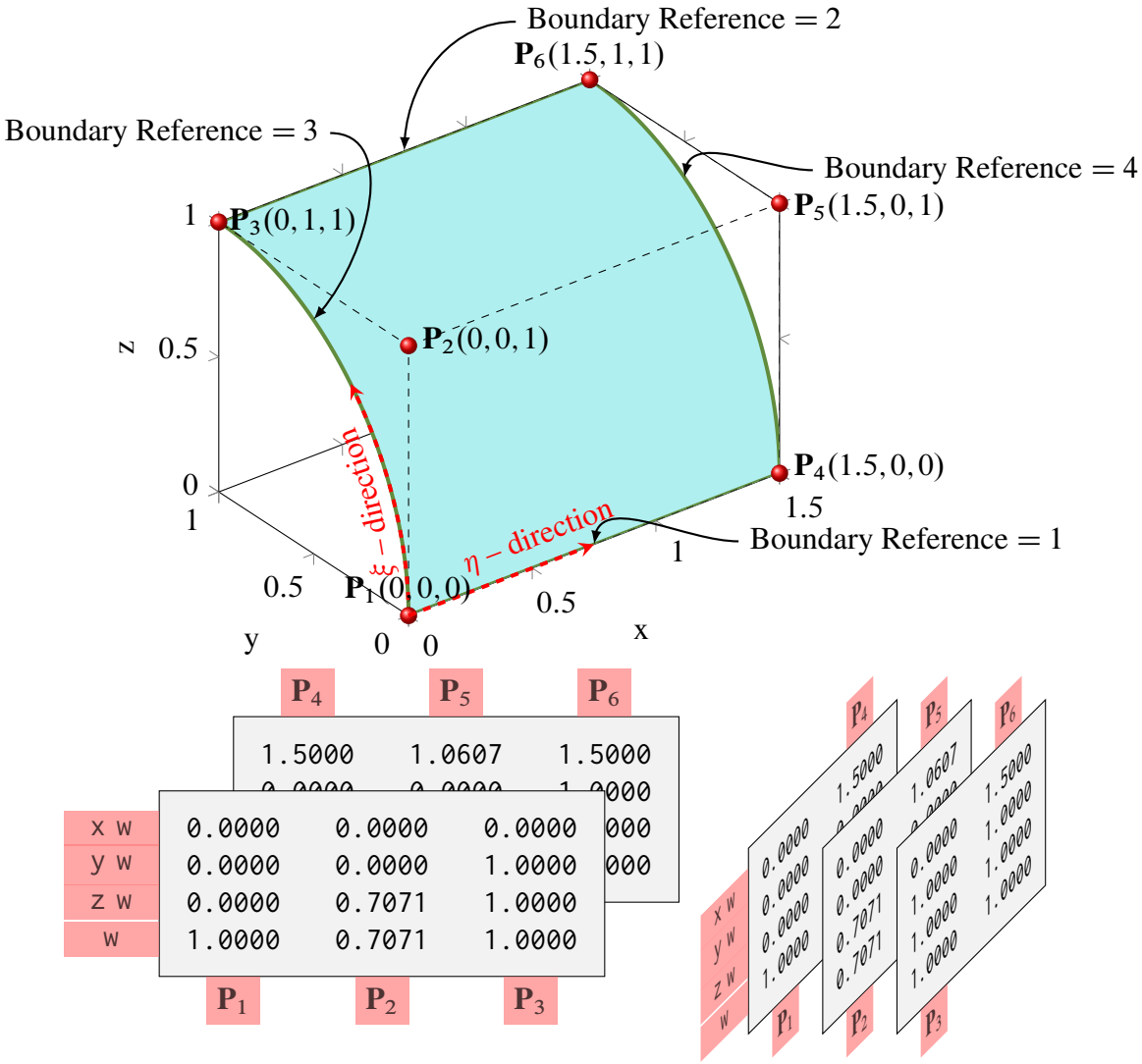
```
radius = 1;
L = 1.5;
% control points for generate a quarter of a cylinder
CtrlPts = zeros(4, 3, 2);

CtrlPts(1 : 3, 1, 1) = [0; 0; 0];
CtrlPts(1 : 3, 2, 1) = [0; 0; radius];
CtrlPts(1 : 3, 3, 1) = [0; radius; radius];

CtrlPts(1 : 3, 1, 2) = [L; 0; 0];
CtrlPts(1 : 3, 2, 2) = [L; 0; radius];
CtrlPts(1 : 3, 3, 2) = [L; radius; radius];

% assign the weights
CtrlPts(4, :, :) = 1;
W = 1 / sqrt(2);

CtrlPts(:, 2, :) = CtrlPts(:, 2, :) * W;
```



(a) Each frontal slice store data for forming the corresponding control polyline in ξ – direction.

(b) Each lateral slice store data for forming the corresponding control polyline in η – direction.

Figure 3.3: Demonstration of data storage for contructing a NURBS surface.

Similar to curve, the ξ and η represent the parameter value from one end of a surface to the other. These identifications play a same role as aforementioned functionality for curve. In addition, these information are very important when we model a multiple patches geometry since it requires two patches must have the same parameter direction at the interface of them (in the case that adjacent patches do not have the same parameter direction, a reverse of direction is needed).

In the Fig. 3.3, "Boundary Reference" is a label which is used to indicate the boundary of the geomtry in SIMO Package. Each label keep an integer value, these values are enumerated based on parameter values. Beginning from ξ and then η direction, the enumeration specified as $\xi = 0$ and $\xi = 1$ corresponds to Boundary Reference = 1 and 2, $\eta = 0$ and $\eta = 1$ corresponds to Boundary Reference = 3 and 4.

– For solid: control point's coordinates are stored in a four-dimensional array where each frontal slice of this array return a three-dimensional array and each frontal slice of this three-dimensional array return a two-dimensional array. So this is an extension of data structure for surface, therefore all conventions are the same as in surface case. An example

code is showed below to illustrate this argument. To avoid confusion, we should fill control point's coordinate to the 4-D array in the order from the first frontal slice `CtrlPts(:, :, :, 1)` to the second frontal slice `CtrlPts(:, :, :, 2)` and so on.

```
r = 2; % inner radius of hollow cylinder
t = 0.5; % thickness of hollow cylinder
h = 6; % height of hollow cylinder

% control points
CtrlPts = zeros(4, 9, 2, 2);

CtrlPts(1 : 3, 1, 1, 1) = [ r; 0; 0];
CtrlPts(1 : 3, 2, 1, 1) = [ r; r; 0];
CtrlPts(1 : 3, 3, 1, 1) = [ 0; r; 0];
CtrlPts(1 : 3, 4, 1, 1) = [-r; r; 0];
CtrlPts(1 : 3, 5, 1, 1) = [-r; 0; 0];
CtrlPts(1 : 3, 6, 1, 1) = [-r; -r; 0];
CtrlPts(1 : 3, 7, 1, 1) = [ 0; -r; 0];
CtrlPts(1 : 3, 8, 1, 1) = [ r; -r; 0];
CtrlPts(1 : 3, 9, 1, 1) = [ r; 0; 0];

CtrlPts(1 : 3, 1, 2, 1) = [ r + t; 0; 0];
CtrlPts(1 : 3, 2, 2, 1) = [ r + t; r + t; 0];
CtrlPts(1 : 3, 3, 2, 1) = [ 0; r + t; 0];
CtrlPts(1 : 3, 4, 2, 1) = [-(r + t); r + t; 0];
CtrlPts(1 : 3, 5, 2, 1) = [-(r + t); 0; 0];
CtrlPts(1 : 3, 6, 2, 1) = [-(r + t); -(r + t); 0];
CtrlPts(1 : 3, 7, 2, 1) = [ 0; -(r + t); 0];
CtrlPts(1 : 3, 8, 2, 1) = [ r + t; -(r + t); 0];
CtrlPts(1 : 3, 9, 2, 1) = [ r + t; 0; 0];

CtrlPts(1 : 3, 1, 1, 2) = [ r; 0; h];
CtrlPts(1 : 3, 2, 1, 2) = [ r; r; h];
CtrlPts(1 : 3, 3, 1, 2) = [ 0; r; h];
CtrlPts(1 : 3, 4, 1, 2) = [-r; r; h];
CtrlPts(1 : 3, 5, 1, 2) = [-r; 0; h];
CtrlPts(1 : 3, 6, 1, 2) = [-r; -r; h];
CtrlPts(1 : 3, 7, 1, 2) = [ 0; -r; h];
CtrlPts(1 : 3, 8, 1, 2) = [ r; -r; h];
CtrlPts(1 : 3, 9, 1, 2) = [ r; 0; h];

CtrlPts(1 : 3, 1, 2, 2) = [ r + t; 0; h];
CtrlPts(1 : 3, 2, 2, 2) = [ r + t; r + t; h];
CtrlPts(1 : 3, 3, 2, 2) = [ 0; r + t; h];
CtrlPts(1 : 3, 4, 2, 2) = [-(r + t); r + t; h];
CtrlPts(1 : 3, 5, 2, 2) = [-(r + t); 0; h];
CtrlPts(1 : 3, 6, 2, 2) = [-(r + t); -(r + t); h];
CtrlPts(1 : 3, 7, 2, 2) = [ 0; -(r + t); h];
CtrlPts(1 : 3, 8, 2, 2) = [ r + t; -(r + t); h];
CtrlPts(1 : 3, 9, 2, 2) = [ r + t; 0; h];
% weights
CtrlPts(4, :, :, :) = 1;
fac = 1 / sqrt(2);
CtrlPts(:, 2 : 2 : 8, :, :) = CtrlPts(:, 2 : 2 : 8, :, :) * fac;
```

These two parameterization data (knot vector(s) and control points) are then collected and putted in a structure data type of MATLAB for later using in computation. To do this in SIMO Package, we need to call the funtion named "CreateNURBS" (`Object = CreateNURBS(KntVect, CtrlPts);`), the return value is a structure array contains the following fields:

- KntVect: knot vector(s) of the B-spline (NURBS) patch.
- uqKntVect: unique knot values of the B-spline (NURBS) patch.
- CtrlPts4D: control point's coordinates in homogeneous space.
- CtrlPts3D: control point's coordinates projected into Cartesian space.
- Weights: weights of control points stored in 1D array.
- Dim: dimension of the B-spline (NURBS) patch (Dim $=$ 1 for curve, Dim $=$ 2 for surface, Dim $=$ 3 for volume).
- NCtrlPts: number of control point in each direction stored as a 1D array.
- Order: order of B-spline (NURBS) patch per each direction stored as a 1D array.
- NNP: number of total control points on the B-spline (NURBS) patch.
- Orientation: orientation of the B-spline (NURBS) patch (sign of Jacobian of mapping), usually in computation we use absolute value of this Jacobian determinant. This infomation is needed in some cases such as to determine the orientation of the normal vector of the boundary, etc.

Basis functions

There are two primary routines implemented in SIMO Package which are intended to evaluate the non-zero values of univariate basis function at parametric points, one is named "BasisFuns.m" and the other is "DersBasisFuns.m", while the former is mainly used in generating geometry, the latter is used in analysis as it simultaneously evaluates the corresponding derivative values of basis functions. The syntax to use these functions is described as follows

- For the routine "BasisFuns.m"

```
N0 = BasisFuns(Idx, Pts, p, KntVect);
```

  Input parameters:

  - `Idx`: span indices of the corresponding parametric points, it can be a single value or an array (the number element of this array have to be equal to the number element of `Pts` array). This array is usually generated by the "FindSpan.m" routine.
  - `Pts`: parametric points at which the values of basis functions are computed
  - `p`: degree of the basis functions
  - `KntVect`: knot vector defining the parameterization

  Output parameter:

  - `N0`: a 2D array which stores evaluated basis functions values of the input parametric points, where each row contains $p + 1$ values computed at a parametric point (as at each parametric point there are always exist $p + 1$ basis functions).

- For the routine "DersBasisFuns.m"

```
N0n = DersBasisFuns(Idx, Pts, p, n, KntVect);
```

  Input parameters:

- `Idx`, `Pts`, `p`, `KntVect`: similar as aforementioned routine.
- `n`: order of derivatives we would like to compute

Output parameter:

- N0n: a 3D array holding values of evaluated basis functions and its derivatives, the size of this array in the third direction depends on the maximum order of derivatives, starting from the zeroth-order derivatives (basis functions) in the first slice, then the first-order derivatives in the second slice, etc., e.g
  - To retrieve the basis functions, we can simply use the command

```
N0 = N0n(:, :, 1);
```

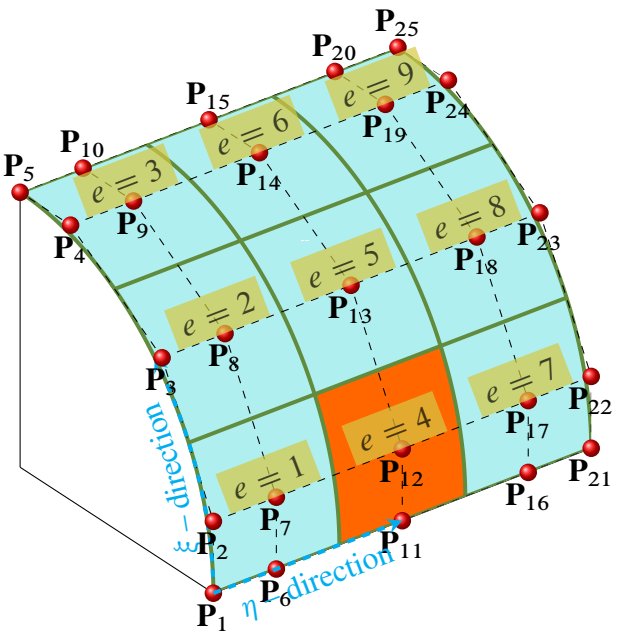
  - To retrieve the first derivatives of basis functions, we invoke

```
N0 = N0n(:, :, 2);
```

3.2 Enumeration and the Local-to-Global Connectivity Array

For the purpose of calculating elemental matrices as well as performing matrix assembly, we need to specify which data for an Isogeometric structured mesh of a patch should be generated. These data are listed below

1. Global control point numbers
2. Element numbers
3. The element-to-control point connectivity array
4. Local control point numbers



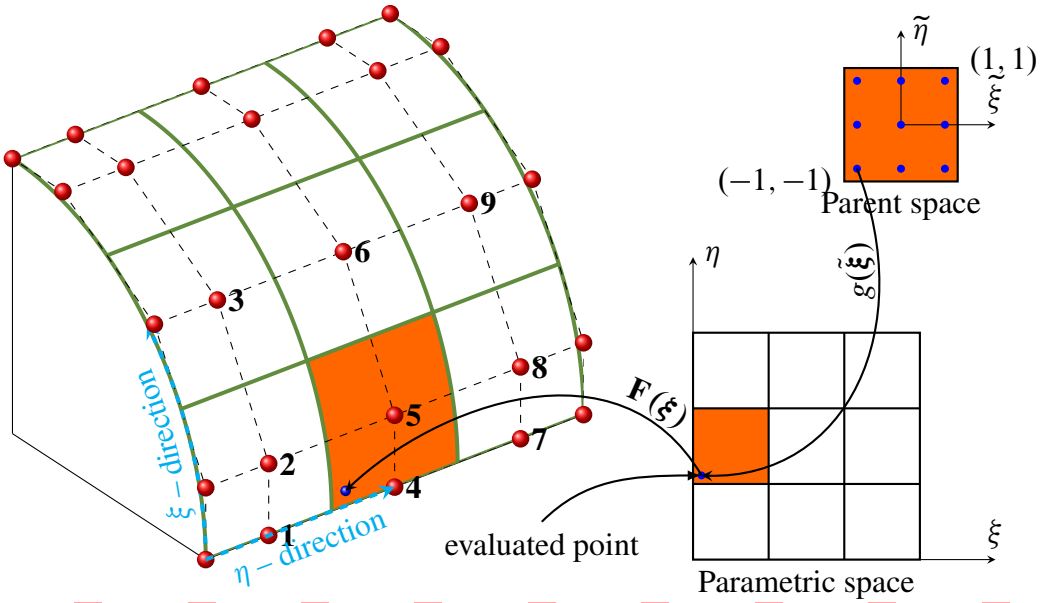
Control Point Connectivities

1th element	1	2	3	6	7	8	11	12	13
2th element	2	3	4	7	8	9	12	13	14
3th element	3	4	5	8	9	10	13	14	15
4th element	6	7	8	11	12	13	16	17	18
5th element	7	8	9	12	13	14	17	18	19
6th element	8	9	10	13	14	15	18	19	20
7th element	11	12	13	16	17	18	21	22	23
8th element	12	13	14	17	18	19	22	23	24
9th element	13	14	15	18	19	20	23	24	25

Figure 3.4: Global control point and element numbers.

In SIMO Package, to take advantage of MATLAB programming and simplify the code, the first two data and the last one are implemented implicitly, that means we do not need to create any MATLAB array to store these information. Let's take an example as depicted in Fig. 3.4. In this example, the NURBS geometry is parameterized by two knot vectors $\boldsymbol{\Xi} = \{0, 0, 0, 1, 1, 1\}$ and $\mathbf{H} = \{0, 0, 1, 1\}$, respectively with $p = 2$ and $q = 1$. For the global control point numbers, the ordering is determined by the indices of the array that stores the control point's coordinates, moving along the ξ direction first (horizontally), followed by η (vertically) (row by row starting at the lower left corner of the parametric space) as in Fig. 3.4. Therefore, we have to follow strictly the procedure as mentioned in the previous section to set up control point's coordinates correctly.

The element numbers are assigned in the same manner as we did for global control point numbers. Due to the fact that only the non-zero knot spans are taken into account in analysis which we call "elements" in IsoGeometric Analysis and we are using open knot vectors that possibly have repeated knot values, some of these knot spans may result in zero interiors. In case of 2D isogeometric structured mesh, the computational domain is parameterized by two knot vectors $\boldsymbol{\Xi}$ and $\mathbf{H}$. The discretization of the physical domain is, therefore, obtained by excluding the multiple knot values in each knot vector. This implementation is incorporated as a member of the NURBS struct which is mentioned before in the Data Representation section, namely "uqKntVect". The number of elements in each direction is then calculated by taking the number of unique knot values minus 1 (`numel(NURBS.uqKntVect)-1;`). Finally, the numbering of elements can be illustrated as in Fig. 3.4.



	1	2	3	4	5	6	7	8	9
R	3.166e-1	7.489e-2	2.139e-3	4.826e-1	1.141e-1	3.256e-3	5.108e-3	1.208e-3	3.450e-5

	1	2	3	4	5	6	7	8	9
$dR/d\xi$	-1.966e+0	1.851e+0	1.150e-1	-2.996e+0	2.821e+0	1.753e-1	-3.172e-2	2.986e-2	1.856e-03
$dR/d\eta$	-2.141e+0	-5.064e-1	-1.446e-2	1.869e0	4.421e-1	1.262e-2	2.720e-1	6.432e-2	1.837e-3

Figure 3.5: Local numbering of the fourth element with basis function values and their first derivatives evaluated at the first Gaussian point of this element.

The purpose of the element-to-control point connectivity array is to scatter local ordering of control points of the element to the global one. For the considering example, the array in question is presented in the right hand side of the Fig. 3.4, where the row indices represent the element numbers and the values stored in each row are global control point numbers pertain to each element. In SIMOPackage, the routine to generate this array is currently implemented in the files named `Mesh1D`

for a 1D patch, `Mesh2D` for a 2D patch and `Mesh3D` for a 3D patch, respectively. For example, to retrieve the connectivity array of the element 4, we invoke

```
Mesh = Mesh2D(Surf, 'VectorField');
ElConn = Mesh.El(4, :);
```

Notice that, this array is solely a control point connectivity array, not a DOFs (Degrees Of Freedom) connectivity array. Naturally, it is a DOFs connectivity array for single field problems, but for multiple fields problems, we have to establish a mapping to relate the control point numbers and their control variables (component values). Let's clarify this matter by investigating a 2D elasticity problem where each control point of this problem has 2 degrees of freedom (2 component values). In this analysis, the final step of the processing stage is to solve the linear algebra system

$$\mathbf{K}\mathbf{d} = \mathbf{F}, \tag{3.1}$$

which require some kind of ordering of component values of control points. There are two types of ordering used widely in practice. The first one is to alternate component values of each control point, the second one is to gather component values which have the same attribute (x or y displacement in this case) into adjacent fragments. These both types are applied for vector $\mathbf{d}$ as follow

$$\mathbf{d} = \begin{Bmatrix} u_{1,x} \\ u_{1,y} \\ u_{2,x} \\ u_{2,y} \\ \vdots \\ u_{n,x} \\ u_{n,y} \end{Bmatrix}, \qquad \mathbf{d} = \begin{Bmatrix} u_{1,x} \\ u_{2,x} \\ \vdots \\ u_{n,x} \\ u_{1,y} \\ u_{2,y} \\ \vdots \\ u_{n,y} \end{Bmatrix}, \tag{3.2}$$

where the left-hand side and the right-hand side represent for the first and the second approach, respectively, the first index i in $u_{i,j}$ identify the control point numbers and the second one j identify the component of the field. Of course, we need to specify the ordering of entries in matrix $\mathbf{K}$ and vector $\mathbf{F}$ as well to adapt to vector $\mathbf{d}$. To do this, we can perform the ordering on the element level by arranging basis functions. We utilize the second approach in SIMOPackage, thus the basis functions matrix and the strain-displacement matrix have the following form

$$\begin{aligned} \mathbf{N} &= \underbrace{\begin{bmatrix} N_1 & N_2 & \dots & N_{n_{en}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & N_1 & N_2 & \dots & N_{n_{en}} \end{bmatrix}}_{2\times 2n_{en}}, \\ \mathbf{B} &= \begin{bmatrix} \dfrac{\partial N_1}{\partial x} & \dfrac{\partial N_2}{\partial x} & \dots & \dfrac{\partial N_{n_{en}}}{\partial x} & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \dfrac{\partial N_1}{\partial y} & \dfrac{\partial N_2}{\partial y} & \dots & \dfrac{\partial N_{n_{en}}}{\partial y} \\ \dfrac{\partial N_1}{\partial y} & \dfrac{\partial N_2}{\partial y} & \dots & \dfrac{\partial N_{n_{en}}}{\partial y} & \dfrac{\partial N_1}{\partial x} & \dfrac{\partial N_2}{\partial x} & \dots & \dfrac{\partial N_{n_{en}}}{\partial x} \end{bmatrix}, \end{aligned} \tag{3.3}$$

the elemental matrix is then scattered to the global one via a DOFs connectivity array adapted from the element-to-control points connectivity array by element-wise adding the total number of control points as

```
for el = 1 : Mesh.NEl
    ElConn = Mesh.El(el, :);
    DOFConn = [ElConn, ElConn + NURBS.NNP];
end
```

For reasons of simplicity, local basis functions (one-to-one correspondence with control points) are numbered in the same manner as the global one (number in the ξ direction first, followed by η) and this process is performed implicitly in SimoPackage. Let's discuss further about this, in order to compute elemental matrices, multivariate basis functions in company with their derivatives are calculated based on tensor product and stored in a 2D array as illustrated in the Fig. 3.5 for the evaluated basis function values and derivatives at the first Gaussian point of the element. Therefore, we can simply control local numbering by putting basis functions values into the array at the corresponding indices. Assuming 2D basis functions and derivatives are need to be calculated for the element `e = sub2ind (Mesh.NElDir, ex, ey)` at the Gaussian point `(qx, qy)`, then this work can be accomplished by employing two nested loops as

```
N0 = zeros(1, Mesh.NEN);
N1 = zeros(2, Mesh.NEN);
for j = 1 : NURBS.Order(2) + 1 % loop over basis functions in eta direction
    for i = 1 : NURBS.Order(1) + 1 % loop over basis functions in xi direction
        N0(k) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 1);
        N1(1, k) = Nx(ex, qx, i, 2)*Ny(ey, qy, j, 1);
        N1(2, k) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 2);
        k = k + 1;
    end
end
```

It is worthy to notice that from the Fig. 3.5, we can know in advance that the determinants of Jacobian matrices of the mapping from parametric space to physical space is a negative number simply because the parametric space need to be swapped in order to compatible with the physical space. This issue is the consequence of the way we set up control points to build the NURBS patch. We almost do not see any affect of this issue in single-patch geometry problems, but it possibly causes strange results in multi-patches geometry cases when one of the patches has negative Jacobian determinants. To avoid this issue, we should always take the absolute value of the Jacobian determinant in the elemental matrices calculation process.

```
% Gradient of mapping from parameter space to physical space
dxdxi = R1*CtrlPts(Mesh.El(e, :), :);

J1 = abs(det(dxdxi));
```

3.3 Elemental Matrices and Assembly Process

Evaluate values of basis functions and their derivatives at quadrature points

In order to perform Gauss integration on the interested physical domain, the calculation to get the values of basis functions and their derivatives at quadrature points is an essential prerequisite. This work is accomplished by a routine which is already implemented in SIMO Package called "calcDersBasisFunsAtGPs.m". It should be noticed that this routine is implemented only for univariate case and the syntax to use it is

```
[Jx, Wx, Xi, Nx] = calcDersBasisFunsAtGPs(p, n, KntVect, d, NGPs, NEl);
```

where the explanations of the input parameters are represented as follows

- `p`: degree of basis functions
- `n`: number of control points
- `KntVect`: knot vector
- `d`: maximum degree of derivatives need to be computed
- `NGPs`: number of gauss points
- `NEl`: number of elements (non-zero knot spans)

and the output parameters are

- `Jx`: jacobian of mapping from parent element to parametric space. These values are stored as a 1D array and the length of this array is equal to the number of the non-zero knot spans
- `Wx`: gaussian quadrature weights stored as a 1D array, the length of it is equal to the number of the Gaussian points
- `Xi`: coordinate of Gaussian points in parametric space
- `Nx`: evaluated basis functions and their derivatives which are stored as an 4-D array, the forth dimension is organized according to orders of derivatives where each slice of this dimension returns a 3-D array which contains data of basis functions (or their derivatives) evaluated at all gauss points in all non-zero knot spans. Some code snippets are represented below as examples for this idea
  - To retrieve $p+1$ evaluated values of $p+1$ basis functions in the **second** non-zero knot span (element) at the **third** Gaussian point, we invoke

```
N0 = Nx(2, 3, :, 1)
```

  - To retrieve $p+1$ evaluated first derivative values of $p+1$ basis functions in the **first** non-zero knot span (element) at the **second** Gaussian point, we invoke

```
N1 = Nx(1, 2, :, 2)
```

For the multivariate cases, basis functions and their derivatives are obtained by using tensor product as mentioned in Chapter 1. To do this in MATLAB, we can simply use for-loop as follows

- For bivariate case

```
N0 = zeros(1, Mesh.NEN);
N1 = zeros(2, Mesh.NEN);
for ey = 1 : Mesh.NElDir(2) % loop over elements in the second direction
    for ex = 1 : Mesh.NElDir(1) % loop over elements in the first direction
        for qy = 1 : NGPs(2) % loop over quadrature points of the second direction
            for qx = 1 : NGPs(1) % loop over quadrature points of the first direction
                k = 1;
                for j = 1 : NURBS.Order(2) + 1 % loop over basis functions
                    for i = 1 : NURBS.Order(1) + 1 % loop over basis functions
                        N0(k) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 1);
                        N1(1, k) = Nx(ex, qx, i, 2)*Ny(ey, qy, j, 1);
                        N1(2, k) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 2);
                        k = k + 1;
                    end
```

```
            end
        end
    end
end
```

- For trivariate case

```
N0 = zeros(1, Mesh.NEN);
N1 = zeros(3, Mesh.NEN);
for ez = 1 : Mesh.NElDir(3) % loop over elements in the third direction
    for ey = 1 : Mesh.NElDir(2) % loop over elements in the second direction
        for ex = 1 : Mesh.NElDir(1) % loop over elements in the first direction
            for qz = 1 : NGPs(3)
                for qy = 1 : NGPs(2)
                    for qx = 1 : NGPs(1)
                        l = 1;
                        for k = 1 : r + 1
                            for j = 1 : q + 1
                                for i = 1 : p + 1
                                    N0(1, l) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 1)*Nz(ez, qz, k,
                                        1);
                                    N1(1, l) = Nx(ex, qx, i, 2)*Ny(ey, qy, j, 1)*Nz(ez, qz, k,
                                        1);
                                    N1(2, l) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 2)*Nz(ez, qz, k,
                                        1);
                                    N1(3, l) = Nx(ex, qx, i, 1)*Ny(ey, qy, j, 1)*Nz(ez, qz, k,
                                        2);
                                    l = l + 1;
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

Up to now we only consider B-spline basis functions which are non-rational basis functions. For visualizing conic geometry as well as its fields (heat distribution, displacement, stresses, etc.), NURBS basis functions are not required since we can readily generate a geometry data in homogenous space using B-splines and then project it to Cartesian space. But for computational aspect, we do need to convert these B-spline basis functions into the NURBS ones by employing the formulation provided in Chapter 1. In SIMO Package, the implementation to perform these procedures is responsible by the routine named "Rationalize.m", the syntax to use it is

- For converting basis functions and their first derivatives

```
[R0, R1] = Rationalize(Weights(Mesh.El(e, :)), N0, N1);
```

- For converting basis functions, their first and second derivatives

```
[R0, R1, R2] = Rationalize(Weights(Mesh.El(e, :)), N0, N1, N2);
```

where the first input parameter is the weights of the control points associated with the input basis functions. This implementation is valid for *univariate, bivariate and trivariate* cases.

Matrix Assembly Process

The usual way to assemble elemental matrices to global matrix is conducted by the following way

```
K = zeros(system_size, system_size);
for el = 1 : NEl
    ElConn = El(el, :);
    . . .
    %compute local stiffness matrix ke
    . . .
    K(ElConn, ElConn) = K(ElConn, ElConn) + ke;
end
```

where the command `K = zeros(system_size, system_size);` is used to allocate a zero global matrix of `system_size`-by-`system_size`. Although this is not a obligatory requirement in MATLAB since MATLAB uses dynamic data structure to hold arrays, it is introduced to avoid reallocating memory for the global matrix during assembly process which can takes a large amount of time, especially if the size of global matrix is relatively large. This approach, however, is only appropriate for small problems (depends on the total RAM amount of your computer) simply because the global matrix is essentially a band matrix in which there are only a few non-zero elements distributed surrounding and along its main diagonal while the remains are zero values. Therefore if this method is applied for medium or large problems, an enormous amount of memory is occupied with useless zero data which predictably lead to an out of memory situation. To tackle this problem, MATLAB provide a data format called "sparse array", where only non-zero values of the array are stored. The above code snippet is need to modified a little bit to take the advantage of this build-in format as

```
K = sparse(system_size, system_size);
for el = 1 : NEl
    ElConn = El(el, :);
    . . .
    %compute local stiffness matrix ke
    . . .
    K(ElConn, ElConn) = K(ElConn, ElConn) + ke;
end
```

where only the command `zeros` is replaced by the command `sparse`. However, this approach can causes degraded performance of the assembly process, this is due to the fact that the sparsity structure of the matrix is internally unknown. Therefore, each time a elemental matrix is scattered to the global one, a reallocation of memory is occurred to hold the non-zero elements of the elemental matrix. This procedure is repeated constantly until the end of the assembly process, which consumes a huge amount of time. This problem can be overcome by pre-calculating the sparsity structure of the global matrix based on mesh structure of the computational domain and then assigning the zero values to the corresponding non-zero value positions of the matrix. Unfortunately, MATLAB does not provide any utility to do that.

For the relatively medium problems, the optimal solution to get rid of all these issues is to employ the so called "'triplet sparse storage" which is officially supported by MATLAB. Follow this method, all elemental matrices are computed in advance and assembled in one call to the build-in "sparse" command of MATLAB. To apply this method, three 1D arrays are required to be created (triplet), where the first array is used to store values of all elemental matrices, the second and the third ones are used to stored the corresponding row and column indices of these values in the global matrix. The `sparse` command use these input parameters to assemble elemental matrices to the global matrix by summing values having the same row and column indices. The below code segment is an example to demonstrate this idea

```
Dof = 1; % degree of freedom per control point
```

```
KVals = zeros((NEN * Dof) ^ 2, NEl);
for el = 1 : NEl % Loop over elements (knot spans)
    Ke = zeros(NDofsEl, NDofsEl);
    . . .
    % evaluate local stiffness matrix ke
    . . .
    KVals(:, e) = Ke(:);
end
jtmp = repmat(1 : NEN * Dof, NEN * Dof, 1);
itmp = jtmp';
ii = El(:, itmp(:))';
jj = El(:, jtmp(:))';

rows = ii(:);
cols = jj(:);
vals = KVals(:);
K = sparse(rows, cols, vals);
clear rows cols vals
```

this approach, however, has possibly three drawbacks

- Since it requires to compute all elemental matrices and store them in a 1D array, several duplicate elemental values (the exact number depends on the continuity of the basis function we choose) which will be summed up in the assembly process are stored separately in the memory → waste memory
- It also requires additional memory to stored the two associated indices arrays → waste memory
- Data of the sparse matrix are stored independently of the three input arrays → there is a peak of memory consumption when the assembly process finished, which leads to an out of memory situation.

Due to these drawbacks, this approach which is implemented in the current SIMO Package version is not suitable for large problems.

3.4 Boundary Conditions

Natural boundary condition

In NURBS-based IsoGeometric Analysis, the enforcement of the natural (Neumann) boundary conditions are performed naturally and in a similar way as in standard Finite Element Method. The implemented routine to impose this type of boundary condition is accomplished in the file named "applyNewmannBdryVals.m".The current implementation of this routine in SIMO Package only cover some standard problems including 2D and 3D heat conduction, 2D and 3D elasticity problems, however, users can readily extend it to any interested problems. The syntax to use it is

```
[Vals, GDofs] = applyNewmannBdryVals(NURBS, Mesh, g, Refs, LAB, varargin);
```

where the input and output parameters are explained as follows:

For the input parameters

- NURBS: NURBS structure of the physical domain
- Mesh: Mesh structure of the physical domain

- g: the function defining the boundary which is provided as an anonymous function handle, e.g
  - For a parabola distributed force: `g = @(x, y)-P * (D ^ 2 / 4 - y ^ 2)/ (2 * I);`
  - For a constant pressure: `g = @(x, y)4;`
- Refs: reference index (indices) of the corresponding boundary (boundaries), in case there are more than one boundary, the reference indices are stored in an array
- LAB: label to identify which type of the problem is investigating, valid labels are
  - `'HFLUX'`: Heat Fluxes in thermal problem
  - `'FX'`, `'FY'` or `'FZ'`: distributed force in elasticity problem
  - `'PRESS'`: pressure boundary condition in elasticity problem
- varargin: a specific input parameter used to handle some particular kinds of problems, e.g
  - In annular plane or hollow cylinder problems, geometry is modelled by a patch with an internal interface (Fig. 3.6). This interface is a consequence of coincident control points at the beginning and the ending of the patch in circumferential direction. In order to solve the problem properly, we have to constraint control values of these overlapping control points such that each pair of control values of the corresponding coincident control points have the same value. One simple way to handle this issue is using Master–Slave method. In SIMO package, this method is implemented by global renumbering DOFs of physical domain in which coincident DOFs are numbered by the same indices. The global numbering is stored in an array and then passed as an extra input argument, i.e

```
[FY, YDofs] = applyNewmannBdryVals(NURBS, Mesh, g, Refs, LAB, GNum)
```

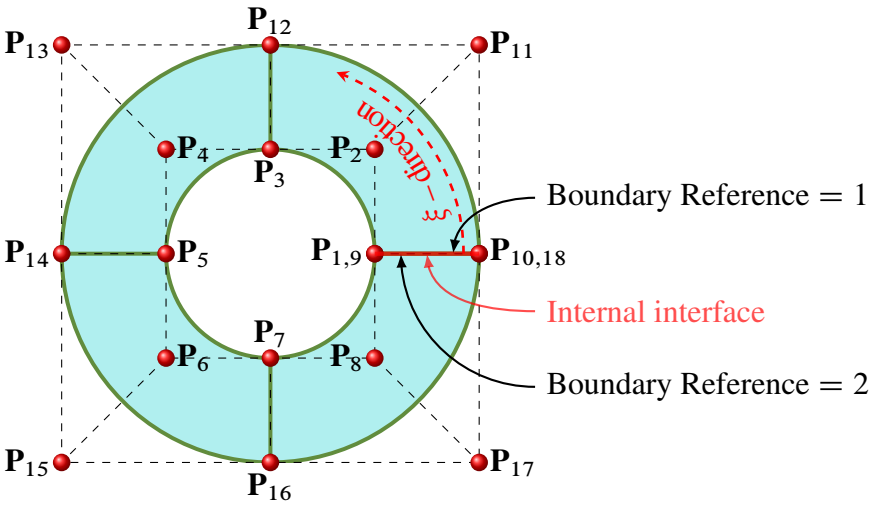


Figure 3.6: An annular patch.

  - For multi-patch problems, the additional parameters are GNum and Boundaries, where GNum is a cell array containing a collection of global numbering arrays of the corresponding patches and Boundaries is a structure data type defining the boundaries of the multi-patch geometry, i.e

```
[FY, YDofs] = applyNewmannBdryVals(NURBS, Mesh, g, Refs, LAB, GNum, Boundaries)
```

The output parameters are

- A force vector **F**: `FY`
- A vector containing the corresponding DOFs: `YDofs`

these force values are then added to the global force vector **F** of the system at the appropriate positions by using the command

```
F(YDofs) = F(YDofs) + FY;
```

Essential boundary condition

Due to non-interpolatory property of NURBS basis functions, a special treatment for applying Dirichlet boundary condition is needed. This issue was first mentioned in the paper [1], in which, for homogeneous Dirichlet boundary condition, boundary values can be directly assigned to control variables. This approach is called direct imposition of Dirichlet BCs, which means we simply assign the prescribed boundary data (evaluated at each control point's position) to the corresponding control variables. In many cases where control points are not located on the boundary, this approach will lead to an incorrect result. For tackling this problem, several methods are proposed such as Lagrange Multiplier Method, Penalty Method, Nitsche's Method, Global Least Squares Fit Method, etc. The current implementation in SIMO Package is based on Global Least Squares Fit Method. This method is presented as follows

Defining u^h is a projection of an arbitrary function $u \in \mathcal{L}^2(\Omega)$ into a finite space of B-spline basis functions $\mathcal{V}^h \in \mathcal{L}^2(\Omega)$, where $\Omega \in \mathbb{R}^d$ is the enforced Dirichlet domain which has dimension d. We need to determine the coefficients c_I such that

$$u^h = \sum_{I=1}^{n} N_I c_I, \tag{3.4}$$

where N_I is the I^{th} B-spline basis function in $\mathcal{V}^h$ and c_I is the associated coefficient of the I^{th} basis function, the function u^h is an approximation of function u. The projection is chosen in such a way that minimize the error produced when projecting u to $\mathcal{V}^h$

$$J(u^h) = \left\| u^h - u \right\|_{\mathcal{L}^2(\Omega)} \to \min, \tag{3.5}$$

where $\mathcal{L}^2$–norm is defined by

$$\left\| f \right\|_{\mathcal{L}^2(\Omega)} = \left(\int_\Omega \left| f(\mathbf{x}) \right|^2 \mathrm{d}\Omega \right)^{1/2} = \sqrt{\left(f(\mathbf{x}), f(\mathbf{x}) \right)_{\mathcal{L}^2(\Omega)}}. \tag{3.6}$$

Since minimizing that norm of the error is equivalent to minimizing the squared–norm error of the projection, therefore to avoid handling the square root, the problem is rewritten as

$$J(u^h) = \left\| u^h - u \right\|^2_{\mathcal{L}^2(\Omega)} \to \min. \tag{3.7}$$

To determine the optimal function u^h of the functional $J(u^h)$ we need to compute the Gâteaux derivative [1] of $J(u^h)$ at $u^h \in \mathcal{V}^h$ in direction $\delta u^h \in \mathcal{V}^h$

$$\begin{aligned} D_{\delta u^h} J(u^h) &= \frac{\mathrm{d}}{\mathrm{d}\alpha} \left(u^h + \alpha \delta u^h - u, u^h + \alpha \delta u^h - u \right) \\ &= 2 \left(u^h + \alpha \delta u^h - u, \delta u^h \right) \Big|_{\alpha=0} \\ &= 2 \left(u^h - u, \delta u^h \right), \end{aligned} \tag{3.8}$$

[1] Gâteaux derivative of inner product $D_h(f,g) = (f, D_h g) + (D_h f, g)$.

the optimum condition occurs when first derivative in all direction is vanished, leads to

$$\left(u^h - u, \delta u^h\right) = 0, \quad \forall \delta u^h \in \mathcal{V}^h, \tag{3.9}$$

or equivalent with

$$\left(u^h, \delta u^h\right) = \left(u, \delta u^h\right), \quad \forall \delta u^h \in \mathcal{V}^h. \tag{3.10}$$

By approximating u^h and δu^h using basis functions, one obtain

$$u^h = \sum_{I=1}^{n} N_I c_I = \mathbf{N}\mathbf{c}, \quad \delta u^h = \sum_{J=1}^{n} N_J \delta c_J = \mathbf{N}\delta\mathbf{c}, \tag{3.11}$$

where $\mathbf{N}$ is a row vector containing basis functions evaluated at boundary, $\mathbf{c}$ and $\delta\mathbf{c}$ are column vectors containing coefficients of the approximation and weight functions, respectively. Substituting 3.11 to 3.10 lead to the system of discretized equations as follow

$$\int_\Gamma \mathbf{N}^{\mathrm{T}}\mathbf{N}\,\mathrm{d}\Omega\mathbf{c} = \int_\Gamma \mathbf{N}^{\mathrm{T}} u\,\mathrm{d}\Omega\,, \tag{3.12}$$

which including a global mass matrix in the left hand side and a force vector in the right hand side. Solving this system of equations we obtain our desired coefficient vector $\mathbf{c}$.

This method is implemented in the file named "projDrchltBdryVals.m" for both single patch and multi-patch problems with the supports for both 2D and 3D cases. In the current version, the implementation this routine is limited to some basic types of problems that involved elasticity, head conduction and plate problems. The input parameters are described below

- NURBS: NURBS structure of the physical domain
- Mesh: Mesh structure of the physical domain
- h: the function defining the boundary which is provided as an anonymous function handle, e.g
  - For homogeneous boundary condition: `h = @(x, y)0`
  - For a specific boundary function: `h = @(x, y)2*x + 3*y`
- Refs: reference index (indices) of the corresponding boundary (boundaries), in case there are more than one boundary, the reference indices are stored in an array
- LAB: label to identify which type of the problem is investigating, valid labels are
  - `'TEMP'` is an abbreviation of "Temperature" and it is used in head conduction problems
  - `'UX'`, `'UY'` or `'UZ'` are labels identifying displacement components and they are used in elasticity problems
  - `'PLATE'` used for applying simply support boundary condition in plate problems
- varargin: an additional input parameter used to handle a NURBS patch having an internal interface and to manage multi-patch problem as mentioned before for natural boundary conditions, e.g
  - For NURBS patch having an internal interface

```
[BdryVals, BdryIdcs] = projDrchltBdryVals(NURBS, Mesh, h, Refs, LAB, GNum)
```

– For multi-patch problems

```
[BdryVals, BdryIdcs] = projDrchltBdryVals(NURBS, Mesh, h, Refs, LAB, GNum, Boundaries
    )
```

The output parameters are listed below

- A coefficients vector $\mathbf{c}_I$: `BdryVals`
- A vector containing the corresponding DOFs: `BdryIdcs`

these parameters are then passed into a routine named "applyDrchltBdryVals.m" which is designed to modify the linear system of equations using partition method, the syntax is

```
[f, d, FreeIdcs] = applyDrchltBdryVals(BdryIdcs, BdryVals, K, f);
```

The linear system is finally solved by invoking the command

```
d(FreeIdcs) = K(FreeIdcs, FreeIdcs) \ f(FreeIdcs);
```

3.5 Visualization

3.6 Bibliography

[1] T. Hughes, J. Cottrell, and Y. Bazilevs, "Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 39–41, pp. 4135–4195, 2005.

CHAPTER 4

SIMULATION EXAMPLES