T-MESHES AND T-SPLINE MODELLING FOR FREE-FORM SHAPE REPRESENTATIONS AND ISOGEOMETRIC ANALYSIS

Alibek Faiznur, B. Eng

Submitted in fulfilment of the requirements for the degree of Master of Science in Mechanical Engineering

School of Engineering
Department of Mechanical Engineering
Nazarbayev University

53 Kabanbay Batyr Avenue,
Astana, Kazakhstan, 010000

Supervisor: Konstantinos Kostas

Co-Supervisor: Luis R. Rojas-Solórzano

January, 2017

DECLARATION

I hereby, declare that this manuscript, entitled "*T-meshes and T-spline Modelling for free-form shape representation and Isogeometric Analysis",* is the result of my own work except for quotations and citations which have been duly acknowledged.
I also declare that, to the best of my knowledge and belief, it has not been previously or concurrently submitted, in whole or in part, for any other degree or diploma at Nazarbayev University or any other national or international institution.

Name: Alibek Faiznur
Date:

Abstract

Nowadays, Isogeometric Analysis has become an effective alternative to traditional Design to Analysis process which uses the same geometry standard both for Design and Analysis purposes. However, NURBS, currently used in Isogeometric Analysis standard, has the limitation of rectangular topology, which results in many superfluous control points. Such points increase the computation time of simulation processes. Moreover, complex object models are composed of a number of NURBS patches, and thus, the designer has to spend time to revise the connectivity between the surfaces. Recently, T-splines surface was explored to overcome the limitations of NURBS. This paper provides the analysis of T-spline implementation in Isogeometric Analysis. It starts from mathematical definition of NUBRS and T-splines, then it describes implementation of T-splines in Rhino and provides the developed program which defines T-mesh parameters.

Acknowledgements

First of all, I would like to express my deep sense of gratitude towards my supervisors, Prof. Konstantinos Kostas and Prof. Luis R. Rojas-Solórzano, who gave me an opportunity to work on this project. I am very grateful to Prof. Kostas for his valuable guidance, encouragement and helpful advices, without his assistance, this work would not have been completed. Also, I would like to thank my Co-Supervisor, Prof. Luis R. Rojas-Solórzano, for his support and inspiration. Finally, I would also like to thank all the faculty and staff members of Nazarbayev University for their cooperation and work in a friendly atmosphere.

Table of Contents

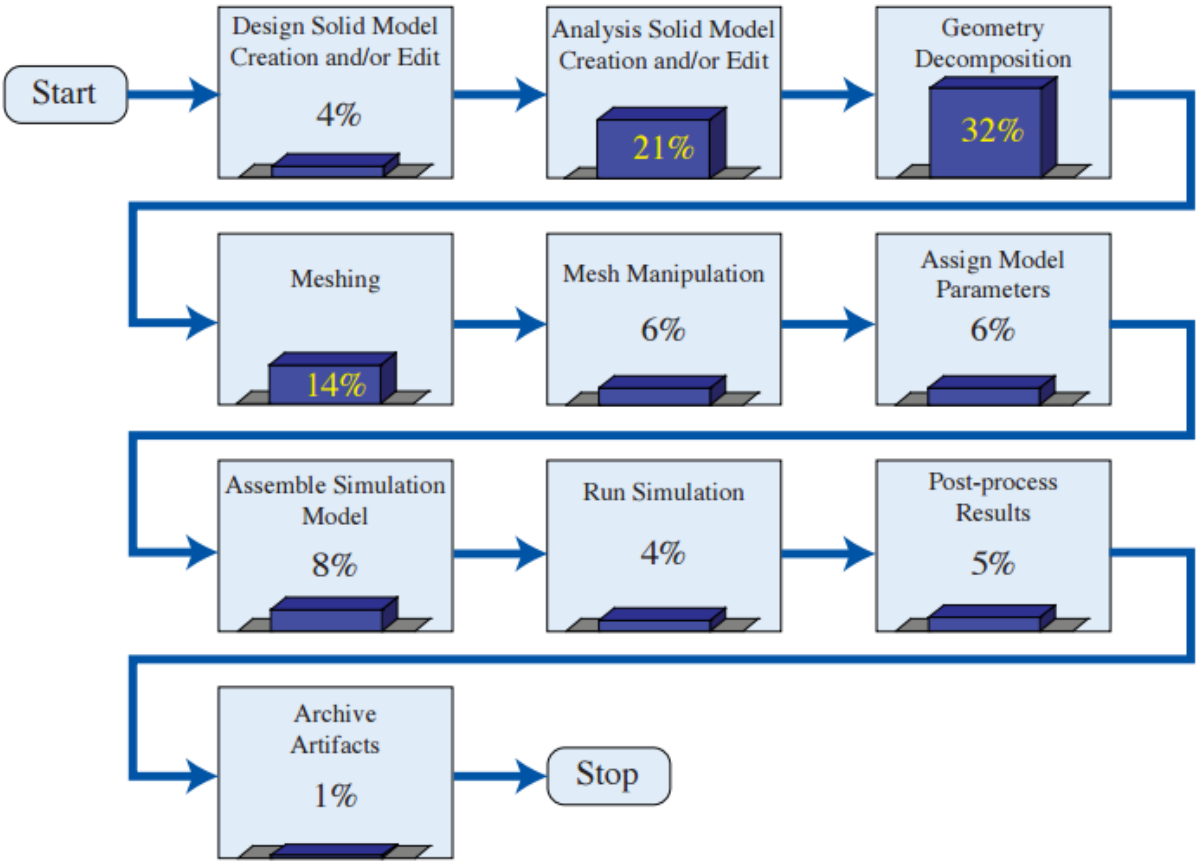
List of Tables

List of Figures

Chapter 1 – Introduction

Currently, Computer Aided Design systems are being widely used by designers all around the world. CAD systems allow the engineer to design and build a model of any product ranging from small ones as bolts and nuts to huge and complex, as submarines and skyscrapers. Moreover, modern analysis and simulation software allows the engineer to modify and optimize the product design. Such tools use a various number of methods and simulations to predict the product properties, including structural mechanics, fluid dynamics, heat transfer, and many others. However, in order to use the simulation tools, it is firstly necessary to translate the product model into analysis suitable geometries. But this is a very complex task which includes a number of time consuming steps [1]. As Hughes and Evans [2] state, translation of complex product model is estimated to take over 80% of the overall analysis time. Figure 1.1 represents results of the study performed at Sandia National Laboratories to analyze the anatomy of the Design to Analysis process and identify time devoted to each step [3].

Figure 1.1: Design to Analysis process [3]



The results have identified that creation of the analysis suitable geometry and adding mesh takes up to 77% of the overall analysis time, while the actual analysis is performed within only 23% of the time. This study clearly shows the inefficiency of conventional Design to Analysis process during which the product model is first created, then its structure is being transformed to suitable for analysis geometry and then the actual simulation starts.

To skip the most time consuming step (geometry conversion), Isogeometric analysis concept was introduced by Hughes, Cottrell, and Bazilevs in 2005 [4]. NURBS standard was first used to design and build the model and employed

directly in the Finite Element Analysis application. However, due to the limitations of NURBS geometry that will be described in detail in the next section, alternative representations, such as T-splines are employed in Isogeometric Analysis. Currently, commercial software tools, in the form of CAD package plug-ins, provide some basic support for T-splines modelling. Unfortunately, their API (Application's Programming Interface) is either limited or not existent, which hinders the use of T-spline modelling in research works. The aim in this thesis is to develop an open, T-spline modelling tool, which will equip researchers with a flexible T-spline modelling solution. The paper starts from mathematical definition of NURBS and T-splines, and then it moves to the description of the developed script and provides results of its operation. Finally, the report provides conclusions and future work suggestions.

Chapter 2 – Geometry Representation in CAD

2.1 B-Spline Curve

Basis spline curve was first investigated in the nineteenth century by N. Lobachevsky [5]. B-spline overcomes the major limitations of Bezier curve, which are fixed curve degree and no local control of the curve's shape. In B-splines each control point is associated with its own basis function and affects the curve shape over a range of parameter values. B-spline curve is a linear combination of control points (p_i) and basis functions. B-spline curve is defined as:

$$r(u) = \sum_{i=0}^{n} p_i \, N_{i,k}(u) \tag{2.1}$$

Where, $N_{i,k}(u)$ is the k[th] order B-spline basis function defined by "Cox-DeBoor" recursion formula as:

$$N_{i,0}(u) = \begin{cases} 1, & u_A \le u < u_{A+1} \\ 0, & otherwise \end{cases} \tag{2.2}$$

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

The basis function $N_{i,k}(u)$ is defined on a knot vector. Knot vector is a sequence of parameter values (knots) in ascending order that determines how the control points affect the curve. For example, [0,0,0,1,2,3] is a valid knot vector, but

[0,0,1,0,2,3] is not. Knot values do not play any role, only the interval ratios to each other are important. In other words, knot vectors [1,1,1,2,3] and [10,10,10,20,30] will result in the same curve. The number of knots equals to the sum of the number of control points (n+1) and the curve order (k). A knot vector is represented by:

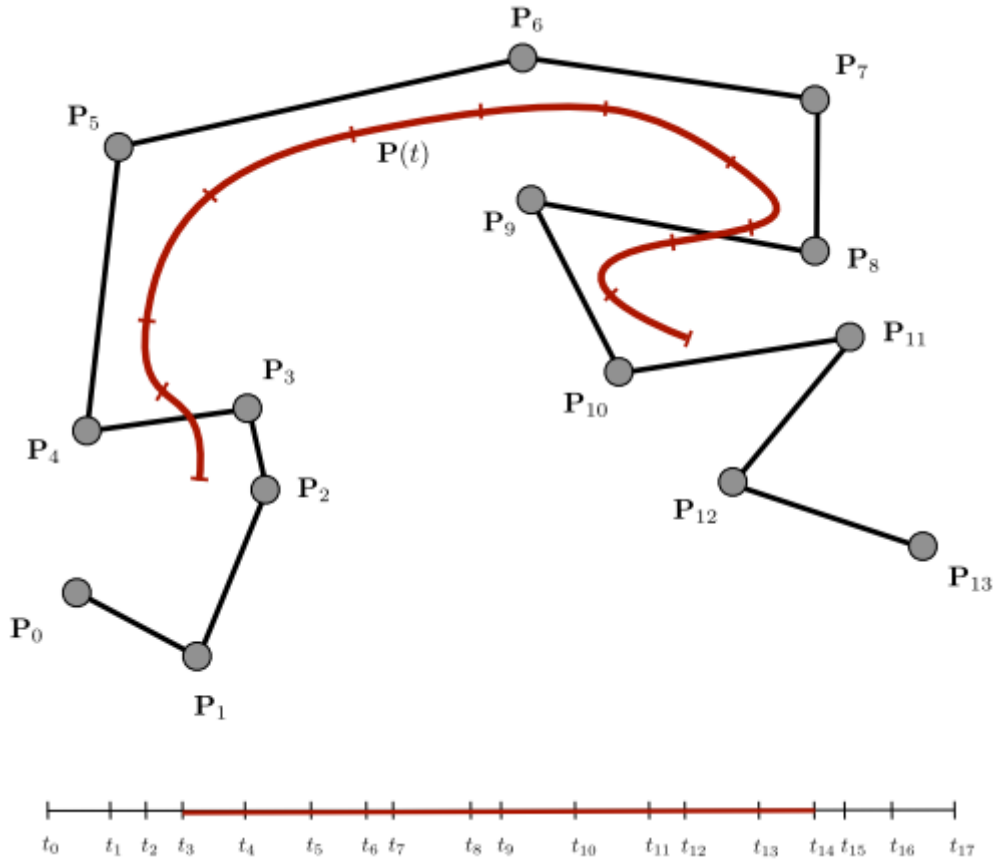
$$U = [u_0, u_1, \dots u_{k+n}] \tag{2.4}$$

The knot vector can be classified as:

- Uniform: constant knot intervals. E.g., [0,1,2,3,4,5]
- Open Uniform (also called clamped): have k identical knots at the ends. E.g., [0,0,0,1,1,1]
- Non-uniform: non-constant knot intervals. E.g., [0,1,3,6,10]

Figure 2.1 demonstrates a B-spline curve created with 14 control points and non-uniform knot vector [6].

Figure 2.1: B-spline Curve [6]



As [7] states, B-spline curves have the following important properties:

- Affine invariance, which include translational, rotational and scaling invariance. In other words, translation and rotation of the control polygon will not change the shape of the curve. And if the control polygon will be scaled to produce a new curve, the result will be the same as if the old curve was scaled.
- Convex hull property: the curve lies within the convex hull of its control polygon.
- Locality: moving a control point or knot affects only the corresponding part of the curve

- Continuity: B-spline curve is C^{d-1}continuous at the segments joints and C^{∞} elsewhere.
- Variation Diminishing Property: a straight line cannot intersect the curve more times than its control polygon.

2.2 NURBS

NURBS stands for Non Uniform Rational B-Splines. In which 'Non-Uniform' term indicates the curve parametrization (knot vector). 'Rational' refers to the fact that NURBS basis functions are rational. 'B-splines' are the parametric curves described earlier. Mathematically, a NURBS curve is defined as:

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,k} w_i P_i}{\sum_{i=0}^{n} N_{i,k} w_i} \tag{2.5}$$

Where: P_i – are the control points; $N_{i,k}$ – B-spline basis functions of order k; w_i stands for the weight of each control point. We can equivalently write:

$$C(u) = \sum_{i=0}^{n} R_{i,k} w_i P_i, \tag{2.6}$$

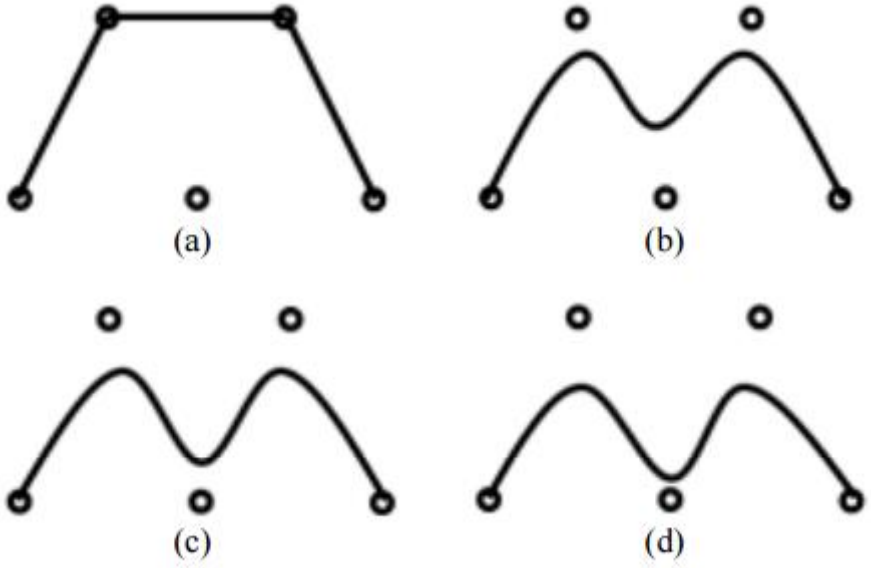
Where

$$R_{i,k} = \frac{N_{i,k}}{\sum_{j=0}^{n} N_{j,k} w_j} \tag{2.7}$$

is the rational NURBS basis function. The control points define the shape of the NURBS curve and moving any of the control points affects the curve locally, as in the case of B-Splines. The magnitude of the control point influence on the curve shape is determined by the weighting factors w_i. Figure 2.2 illustrates how

assigning different weights to the third control point of a degree 2 NURBS curve affects its shape.

Figure 2.2: Weighting factor effect [8]



In case (a) illustrated in Figure 2.2, the weight of the 3rd control point is set to 0. Thus, this point has no influence on the curve shape. In cases (b), (c), and (d), the weighting factor is set to 0.5, 1, and 2, respectively. It can be observed, that as the weighting factor value increases, the effect of the control point also increases.

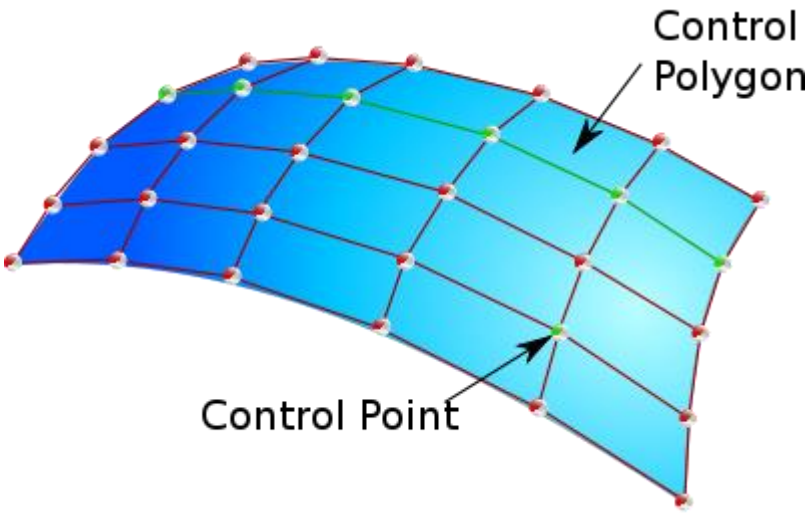
NURBS surface is a parametric tensor product of curves defined by the equation:

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,k}\,(u)N_{j,q}(v)w_{i,j}P_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,k}\,(u)N_{j,q}(v)w_{i,j}} \qquad (2.8)$$

where k and q are the orders of the curves in the two directions

Figure 2.3 illustrates an example of NURBS surface.

Figure 2.3: NURBS Surface [9]



As it was stated in the Introduction, currently, NURBS are being widely used in the Computer Aided Design industry. As stated in [9], any process from animation and illustration to manufacturing can be implemented using NURBS models. As [10] states, the reasons of NURBS popularity are the following:

- It provides a unified mathematical basis for designing standard analytic shapes, like conics and quadrics, and free-form shapes, such as human faces.
- NURBS has a local modification property, thus provides flexibility and large variety of shapes
- NURBS algorithms are computationally stable and fast
- NURBS are invariant under rotation, translation, scaling and parallel and prospective projection.
- NURBS are generalizations of non-rational B-splines as well as rational and non-rational Bezier curves and surfaces.

However, there are some limitations in NURBS representation. As [11] states, the major limitation is that NURBS control points must lie topologically on a rectangular grid. Which means that typically, a significant portion of control points is added to the model just to satisfy rectangular topology. Another limitation of NURBS is that it has no local refinement option. Suppose that we have a mesh as the one shown in Figure 2.4 and we want to refine it as it is shown in Figure 2.5.

Figure 2.4: Control Mesh

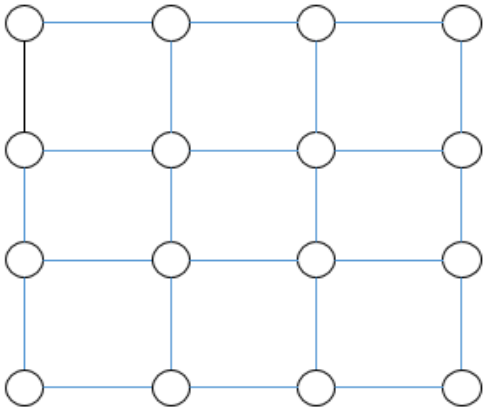
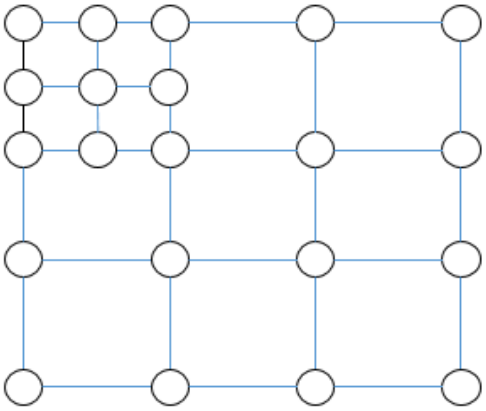
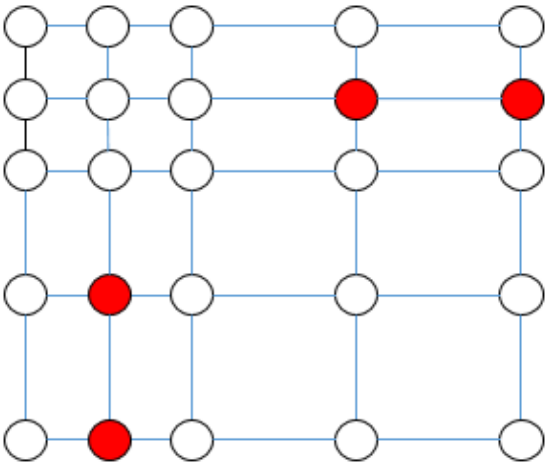


Figure 2.5: Refinement Process



In NURBS, the result of such refinement will be the mesh provided in Figure 2.6.

Figure 2.6: NURBS Refinement Result

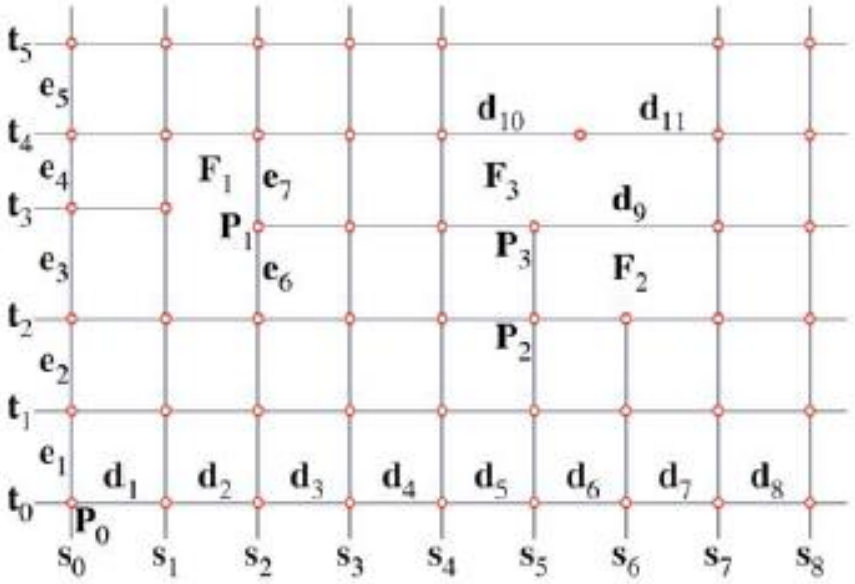


The refinement in the NURBS case will result in creation of unnecessary control points, colored red in Figure 2.6. These control points are added to the mesh due to topological constraint of the NURBS. Superfluous control points increase the complexity of the model and, consequently, the time required to analyze the model or run the simulation. Moreover, superfluous control points may cause unwanted ripples on the model surface, so the designer will spend time to remove those ripples and rectify the shape. To minimize the number of superfluous control points, T-spline representation was developed.

2.3 T-Splines

As Sederberg states in [12], the main difference between T-spline and B-spline (or NUBRS) is that the control grid for T-splines, called T-mesh, allows T-junctions, which has the form of ├, ┴, ┬ or ┤, while the control grid of B-splines does not. T-junctions are included in the T-mesh example depicted in Figure 2.7.

Figure 2.7: T-mesh [12]



Control points P_1 and P_3 are positioned at T-junctions. Similarly to B-splines, knot intervals are assigned to all edges and are used to express knot information for T-splines. But in a T-mesh knot vectors are defined locally and there's not a single knot vector for each direction. Knot intervals must obey the rule that the sum of all knot intervals along one side of any face must be equal to the sum of knot intervals of the opposite side. For example, for the T-mesh depicted in Figure 2.7, for Face 1 we need to have: e3+e4=e7+e6. Similarly, for faces 2 and 3, d9=d6+d7 and d9+d5=d10+d11 need to hold, respectively

Using the knot intervals assigned, it is possible to define a local knot coordinate system. The first step is to choose the control point that will be the origin for parameter domain (s,t) = (0,0). Then, for each vertical edge assign a 's' knot value, and for each horizontal edge assign a 't' knot value. As it can be inferred from Figure 2.7, P_0 (s_0,t_0) was set to be the origin, and s_i and t_i was assigned to each vertical and horizontal edges, respectively. So, the knot coordinates for P_1 are (s_2, $t_2 + e_6$), for P_2 are (s_5, t_2), and (s_5, $t_2 + e_6$) are the knot coordinates of P_3.

T-spline surface explicit formula is:

$$P(s,t) = \frac{\sum_{i=1}^{n} P_i B_i(s,t)}{\sum_{i=1}^{n} B_i(s,t)} \tag{2.9}$$

Where: $P_i = (x_i; y_i; z_i; w_i)$ stands for control points in homogeneous coordinates having the weight w_i as the fourth coordinate and $\frac{1}{w_i}(x_i, y_i, z_i)$ are the

corresponding Cartesian coordinates. The Cartesian coordinates of pointson the surface are defined, therefore, as:

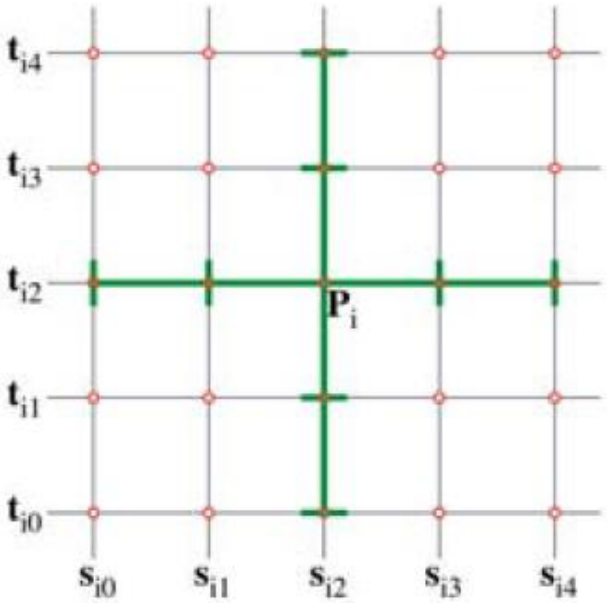
$$\frac{\sum_{i=1}^{n}(x_i, y_i, z_i)B_i(s,t)}{\sum_{i=1}^{n} w_i\, B_i(s,t)} \tag{2.10}$$

where $B_i(s,t)$ is the blending function given by:

$$B_i(s,t) = N[s_{i0}, s_{i1}, s_{i2}, s_{i3}, s_{i4}](s)N[t_{i0}, t_{i1}, t_{i2}, t_{i3}, t_{i4}](t) \tag{2.11}$$

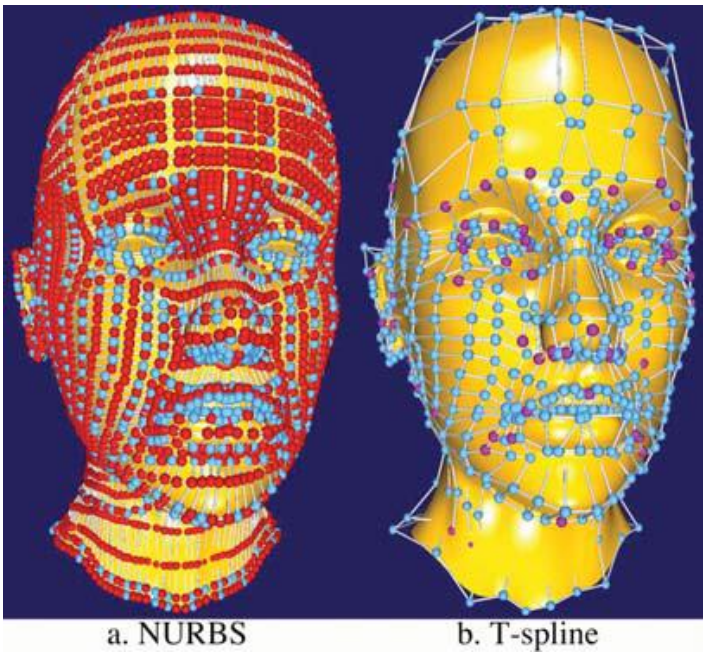
Where $N[s_{i0}, s_{i1}, s_{i2}, s_{i3}, s_{i4}]$ and $N[t_{i0}, t_{i1}, t_{i2}, t_{i3}, t_{i4}](t)$ are, in this case, cubic B-spline basis functions associated with s_i and t_i knot vectors, respectively, as illustrated in Figure 2.8.

Figure 2.8: Knot lines for basis Function [12]



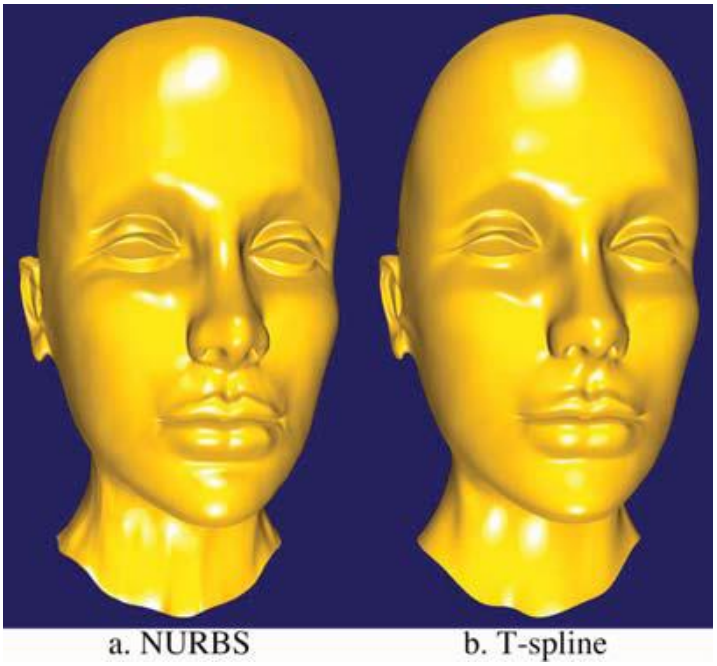
The main advantage of T-spline surface is that a model can be represented with much less control points when compared with NURBS. Figure 2.9 illustrates the same human head modeled with NURBS and T-spline surfaces.

Figure 2.9: Human Head Model [12]



The NURBS model, in the figure above, has 4712 control points, while the T-spline one has only 1109 control points, which means that more than 75% of the control points in the particular NURBS model are superfluous (red colored ones). Figure 2.10 illustrates the appearance of the models.

Figure 2.10: Head Model [12]

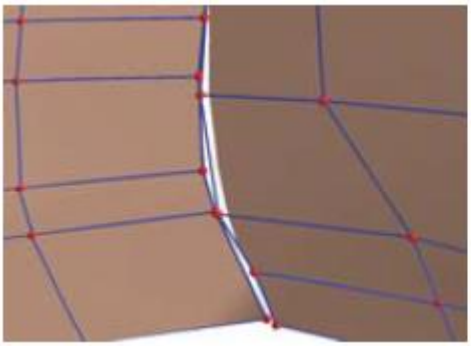


As it can be observed from Figures 2.9 and 2.10, the T-spline surface eliminates the superfluous control points, and, consequently, undesired ripples.

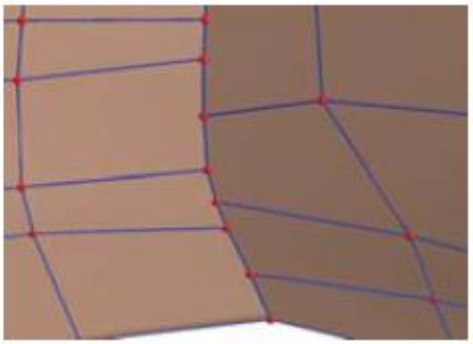
Another advantage of T-splines is that NURBS surfaces with different knot vectors can be merged together into a single T-spline surface. A hand model made of seven B-spline surface patches is shown in Figure 2.11 (a).

Figure 2.11: Hand Model composed of B-spline Surfaces [12]

(a) NURBS hand model



(b) A gap between patches



(c) Patches merged

When a CAD model comprised of several NURBS is being deformed, a gap between neighboring surfaces, indicated by a green rectangle in Figure 2.11 (a), may occur. Thus, the designer has to spend time to eliminate that gap. But using

a T-spline to merge the composing surface patches as illustrated in Figures 2.11 (b) and (c) prevents the formation of such gaps.

Taking into account the advantages of T-splines, it becomes clear that T-splines implementation in computer aided design process offers significant benefits in modelling and analysis.

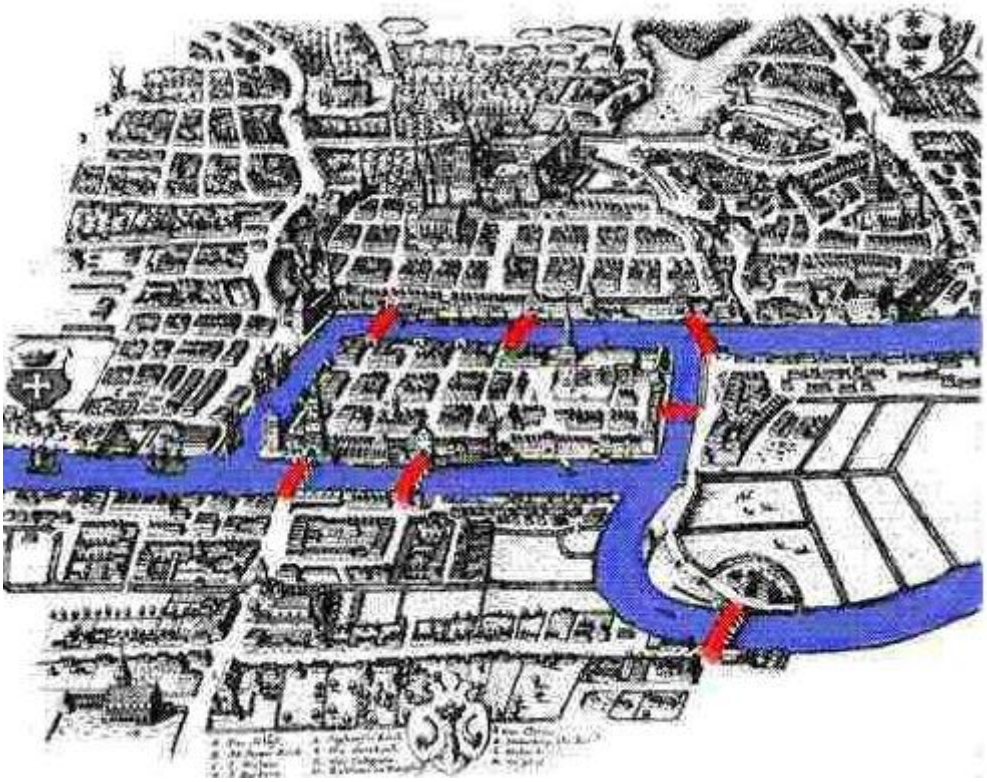
The next section will provide a brief description of basic graph theory terms that will further be used in the report.

Chapter 3- Graph Theory

3.1 Introduction to Graph Theory

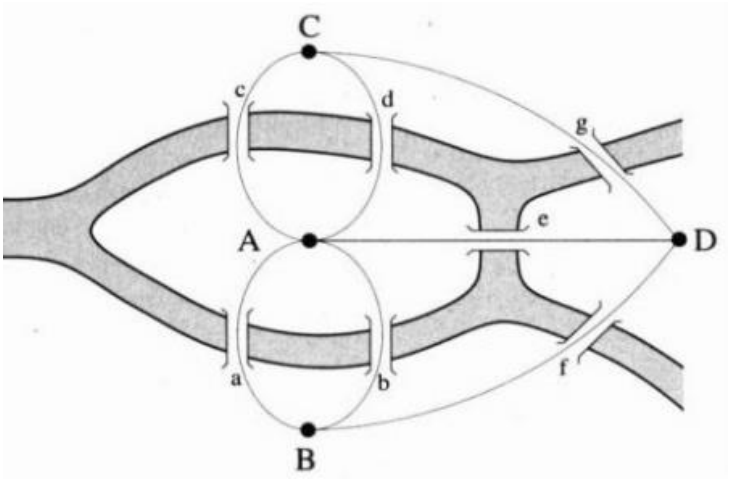
As [13] states, in 18th century Leonard Euler was asked to solve the Königsberg bridge problem, which involved the quest of finding a way of crossing all seven bridges of the city, colored red in Figure 3.1, only once.

Figure 3.1: Königsberg Bridge problem [13]



While solving the problem, Euler has implemented a new approach of addressing the problem, which is shown in Figure 3.2, and laid the foundation of a new branch of mathematics, currently called Graph Theory. Today, Graph Theory is being widely used in many applications, including social network, traffic, navigation, etc.

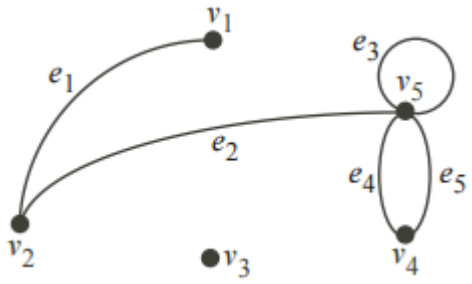
Figure 3.2: First Graph Theory Application [13]



3.2 Graph Theory Basics

A graph is defined as a pair of sets, containing vertices (V) and edges (E), formed by pairs of vertices in V. Mathematically, a graph is defined as $G = (V, E)$, where V is a set of vertices and E ⊆ V×V. Figure 3.3 illustrates a graph formed by V = [v1, v2, v3, v4, v5] and E = [(v1,v2), (v2,v5), (v5,v5), (v5,v4), (v5,v4)] denoted as [e1, e2, e3, e4, e5], respectively.

Figure 3.3: Graph [14]



A graph can be directed or undirected. If for all v and w ∈ V: (v, w) ∈ E and (w, v) ∈ E, then the graph is undirected. Otherwise, the graph is directed. Figures 3.4

(a) and (b), taken from [14], illustrate directed and undirected graphs, respectively.

Figure 3.4 (a): Directed Graph

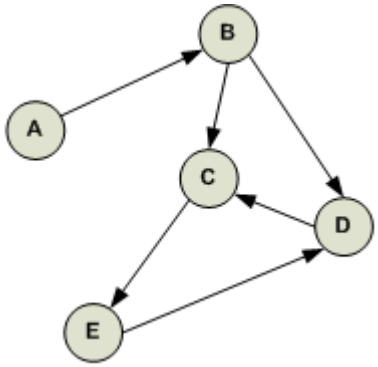
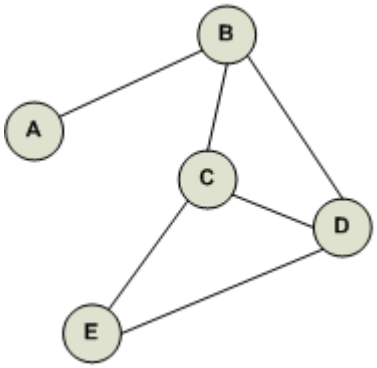


Figure 3.4 (b): Undirected Graph



Also, a graph can be connected or disconnected. If there is a path between each vertex pair, a graph is said to be connected. Otherwise, it is disconnected. The examples of connected and disconnected graphs are provided in Figures 3.5 (a) and (b), respectively.

Figure 3.5 (a): Connected Graph

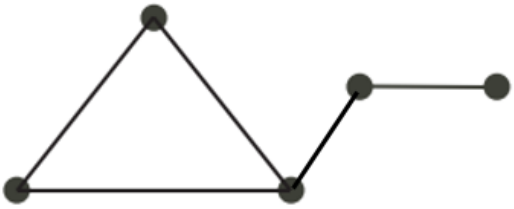
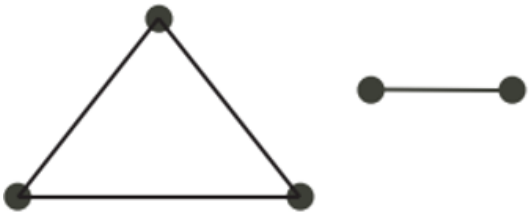


Figure 3.5 (b): Disconnected Graph

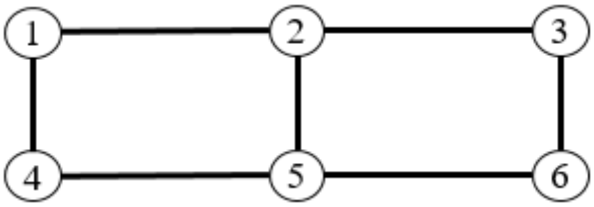


A graph can be represented by the adjacency matrix. If the graph (G) has n number of vertices, the adjacency matrix of G will be n×n matrix $A = (a_{i,j})_{0 \le i,j \le n-1}$ with

$$a_{i,j} = \begin{cases} 1 & if\ there\ is\ an\ edge\ (i,j) \\ 0 & otherwise \end{cases} \quad (3.1)$$

A cycle of the graph is a path that starts and ends at the same vertex with no repetition of other vertices. In the scope of the current project, the cycles are classified into two types: simple and composite cycles. If a cycle can be decomposed into two smaller ones, it is said to be composite, otherwise it is simple. On the graph illustrated in Figure 3.6, cycles 1→2→5→4→1 and 2→3→6→5→2 are simple, but a cycle 1→2→3→6→5→4→1 is composite.

Figure 3.6: Graph Example 1



Chapter 4 – Rhinoceros

4.1 Software Overview

Rhinoceros is a commercial computer aided design package used for 3D modelling. It is developed by Robert McNeel & Associates, an American private company. The software focuses on generating precise shapes of curves, surfaces and solid objects using the NURBS parametric representations. Furthermore, the software package includes a large number of plug-ins that complement and expand the software capabilities in specific fields, including engineering, animation, 3D printing and other. Rhino supports two scripting languages, which are Python and Rhinoscript.

Rhinoscript language was used in the current project. This scripting language is based on the Visual Basic Scripting language (VBScript) developed by Microsoft. Using Rhinoscript, the user can add functionality to Rhino by introducing new functions and automate repetitive steps by introducing loops.

4.2 RhinoScript Basics

We introduce RhinoScript fundamentals, by examining the implementation of the following problem: the final examination grades of the students of a particular class are: 80%, 75%, 51%, 38%, 57%, 83% 69%, 91%, 23%, and 42%. Let’s assume that we want to find the number of students who scored below 40% and failed the examination. To perform this task, the following script could be used:

```
1) Dim scores
2) scores = array(80, 75, 51, 38, 57, 83, 69, 91, 23, 42)
3) Dim i, failed
4) failed = 0
5) For i=0 To ubound(scores)
6)      If scores(i) < 40 Then
7)              failed = failed + 1
8)      End If
9) Next
10) Rhino.Print "Number of Students Failed: " & failed
```

To declare a variable, ‘Dim’ command is used. The script starts from introducing ‘scores’ variable and creating an array with students’ scores. Then, variables ‘i’ and ‘failed’ are declared and the variable ‘failed’ is set to be zero. At the lines 5-9, ‘For…Next’ loop is created in order to pick the scores one by one and check if it is less than 40 by using ‘If …End If’ conditional statement written in the lines 6-8 of the script. When the condition is met, the value of the variable ‘failed’ is being increased by one. Finally, when all the scores are analyzed, the program prints the number of students failed by using the ‘Rhino.Print’ command. The system response will be: “Number of Students Failed: 2”.

The described example demonstrates how the basic RhinoScript commands and loops could be used to perform a given task. Another important point to mention is that, RhinoScript language allows the user to define new functions. The script provided below demonstrates how the problem described earlier could be addressed by introducing a new function.

```
1) Dim scores
2) scores = array(80, 75, 51, 38, 57, 83, 69, 91, 23, 42)
```

```
3) failed_students scores
4)
5) Function failed_students(scores)
6)      Dim i, failed
7)      failed = 0
8)      For i=0 To ubound(scores)
9)              If scores(i) < 40 Then
10)                     failed = failed + 1
11)             End If
12)     Next
13)     Rhino.Print "Number of Students Failed: " & failed
14) End Function
```

Now, the third line of the script calls function ‘failed_students’, defined at the lines 5-14, to operate by using the ‘scores’ variable defined at the lines 1-2.

The next chapter of the report describes the T-spline implementation developed in RhinoScript.

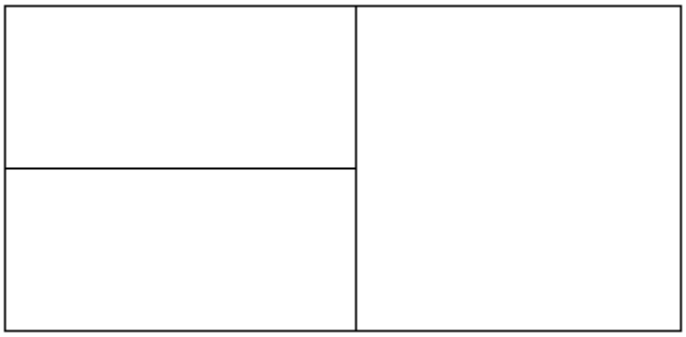
Chapter 5 – T-spline Implementation in Rhinoceros

In the scope of the project, Depth-First search (DFS) algorithm was implemented in Rhino and was used to determine the basic cycles in a graph. Then, to extract all the simple cycles of the graph, another function was developed in Rhino. This function uses the basic cycles identified by DFS, and if the cycle is composite, it decomposes the cycle into two new ones and repeats the operation until it finds all the simple cycles of the graph, which correspond to faces in the T-mesh.

5.1 Script Description

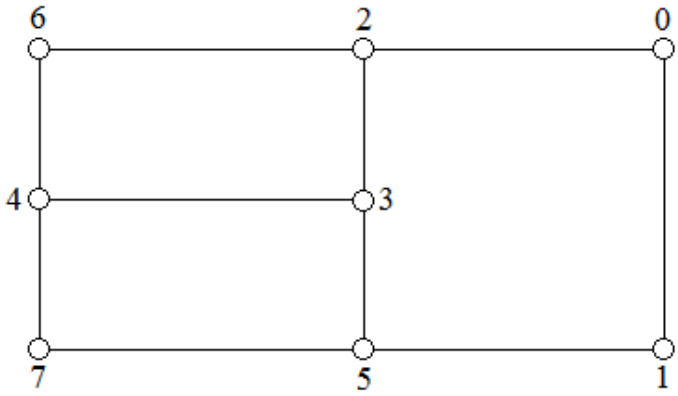
This report section will outline the operation of the developed Rhino script, which is provided in the Appendix A. A very simple T-mesh example illustrated in Figure 5.1 will be used to describe the main steps of the prepared code.

Figure 5.1: Simple T-mesh example



The program takes as input a T-mesh represented with a set of appropriately generated line-segments. The T-mesh is treated as a planar graph and our first step is the creation of its adjacency matrix. The adjacency matrix will store the edge and control points connectivity. The process starts by identifying control points at the intersections of the lines, which is performed by the commands written in the first twenty-five lines of the script (Refer to Appendix A). The result of this step is shown in Figure 5.2.

Figure 5.2: Control Points added



Then, some auxiliary variables and matrices are introduced. Specifically, vertices2edge and edge2vertices are used to store information on which pair of control points defines an edge and which control points are the endpoints of a particular edge, respectively (lines 26-56). Finally, using these variables, the adjacency matrix is being computed and displayed on the software command prompt window (lines 57-70). Figure 5.3 depicts the adjacency matrix corresponding to the T-mesh in Figure 5.2.

Figure 5.3: Adjacency Matrix

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	0	0	1	0	0
2	1	0	0	1	0	0	1	0
3	0	0	1	0	1	1	0	0
4	0	0	0	1	0	0	1	1
5	0	1	0	1	0	0	0	1
6	0	0	1	0	1	0	0	0
7	0	0	0	0	1	1	0	0

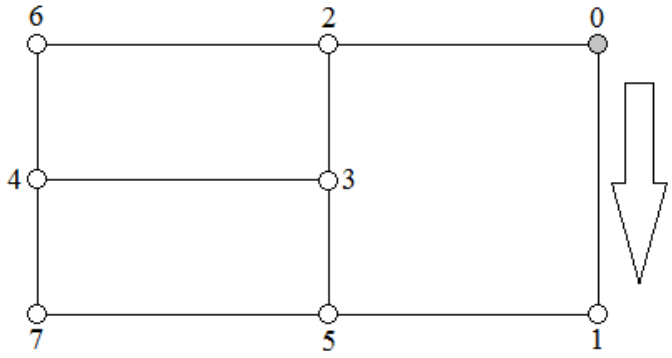
The first row (j index) and first column (i index), in the figure above, correspond to control point indices, while the remaining cells contain a value of 1, if the i control point is connected to the j one, and a 0 value otherwise

Following the generation of the adjacency matrix, the Depth-First Search (DFS) algorithm is used to extract the basic cycles from the given T-mesh graph. The statement in line 74 calls the corresponding DFS function materialized in lines 181-197. The DFS function starts by assigning a “white color” to all control points of the graph. In other words, we consider all the control points as not visited yet. Then, it starts considering points one at a time; if the color of the point in question is white, i.e., the point is not visited yet, it calls DFS-visit function materialized in lines 199-219. When the DFS-visit function is called, firstly it changes the color of the point from white to gray (visited) and then it calls GetConnected function

implemented in lines 221-235, which uses the adjacency matrix to extract the neighboring control points.

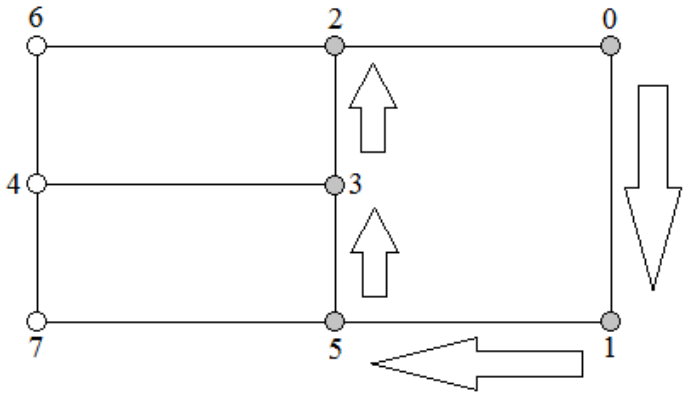
Once the neighboring points are found, the code moves to one of them, makes the previous point a “parent” of the selected one and reruns the DFS-visit function with the new one as argument. In the first call, it will move from point “0” to “1”, as it is shown in Figure 5.4 and make point “0” the parent of point “1”.

Figure 5.4: DFS-visit function



When it arrives to point “1”, it first checks if its’ color is grey, if not, it reruns the DFS-function, changes its color to grey and moves to the next point. This will continue until the time we reach point “2”. At that moment, the graph will be as it is shown in Figure 5.5.

Figure 5.5: DFS at point “2”



At this stage, it will find that the color of point “0” (which is neighboring to 2) is grey and that point “0” is not a parent of point “2”. This signifies that a basic cycle was found.

Then, it will call the ExtractCycle function (lines 238-272) to identify the control points of the cycle and CountCycleEdges function (lines 274-280) to count the number of edges composing the cycle. Finally, the program will print the control points and number of edges of the found cycle, and change the color of starting point from grey to black.

Then, the program will return to the DFS function and repeat the above operations by using another starting point until all points (nodes) are turned to black. Finally, it will print all the cycles found. For this example, the DFS function outputs the following cycles:

Cycle1: 2→3→5→1→0→2 Edge Count: 5

Cycle 2: 4→6→2→3→4 Edge Count: 4

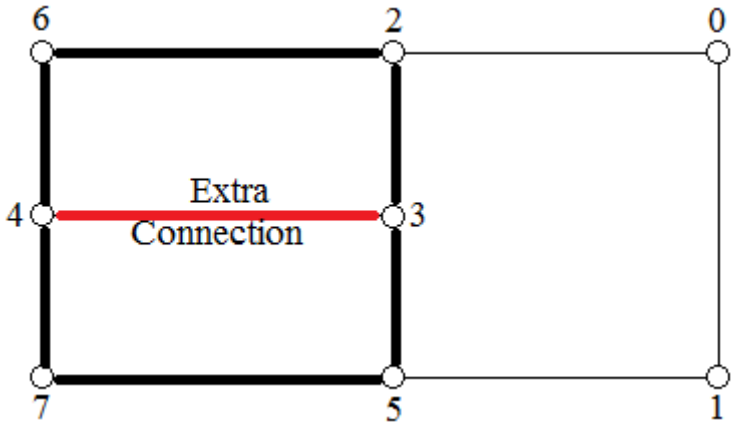
Cycle 3: 7→4→6→2→3→5→7 Edge Count: 6

DFS function ensures that it has covered all the graph points, but it does not guarantee that all the simple cycles are found, because some of the cycles could be composite ones, like Cycle 3 which includes two simple ones: 4→6→2→3→4 and 4→3→5→7→4 which was not found. However, from the identified cycles, we know that cycles with up to 5 edges cannot be composite and hence we insert

those to the list of simple cycles. This step is implemented in lines 77-89 of the script. After that, the CycleType function is executed to find the remaining simple cycles in the graph.

The aim of the CycleType function is to identify whether a basic cycle found by DFS command is composite or simple. If the cycle is composite, CycleType function will divide it into two new cycles. The function uses three different criteria to distinguish composite cycles. The first one is: if any vertex of a cycle has a one-edge connection to another vertex of the same cycle and this edge is not listed in the cycle definition, then the cycle is composite. For instance, in the case of our T-mesh example, and for Cycle 3: 7→4→6→2→3→5→7, point "3" has a connection with point "4", and the corresponding edge is not listed in the cycle path shown in Figure 5.6.

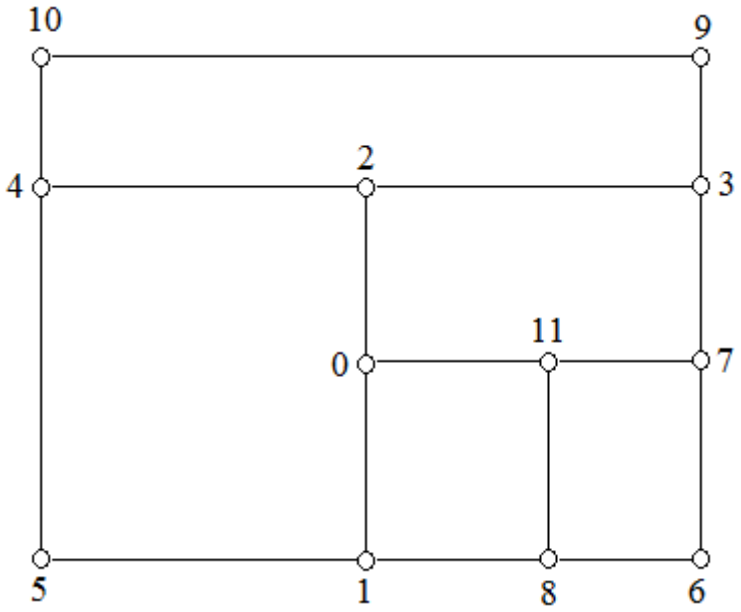
Figure 5.6: Composite Cycle Type 1



In such cases, CycleType function identifies the additional edge between the vertices and marks Cycle 3 as composite. It subsequently divides it into two new cycles. In this case, the new cycles will be: 7→4→3→5→7 and 4→6→2→3→4.

To describe the remaining two criteria of identifying composite cycles, another T-mesh example illustrated in Figure 5.7 will be used.

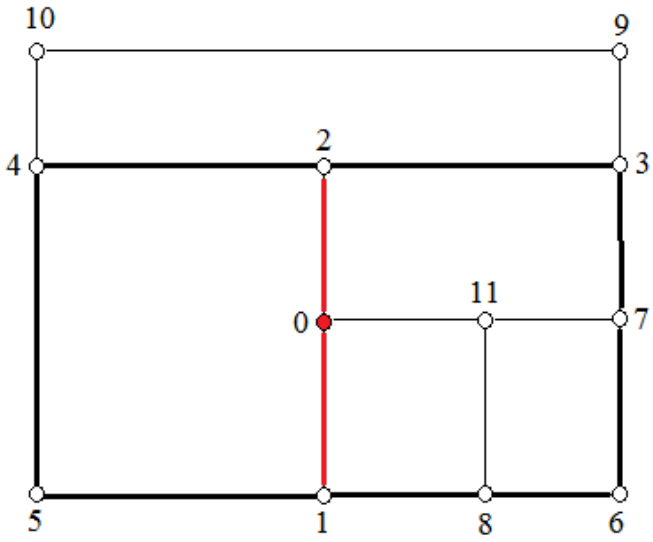
Figure 5.7: T-mesh example



When the DFS function is applied to the graph above, it outputs several cycles, one of which is: 8→6→7→3→2→4→5→1→8 with 8 edges in total. So, when CycleType function starts to analyze this cycle, the first criterion fails to identify the cycle as composite. In this case, the second criterion is applied, which is defined as following: if there is a point, not belonging to the cycle, which is however connected to at least three vertices of the same cycle, the cycle is composite. For our case, such point does not exist, because both points “11” and “0” are connected to only two vertices of the cycle. Thus, we employ an additional third criterion in composite cycles identification.

The third criteria states that, if there is a point on the graph, which is not included in the cycle and has a connection to two vertices of a cycle, the cycle is composite. In this case, the program will first identify that the point “0” is connected to two vertices of a cycle, which are “1” and “2”, as it is shown in Figure 5.8. Then, it marks the cycle as composite and divides it into two new cycles: 8→6→7→3→2→0→1→8 and 2→4→5→1→0→2.

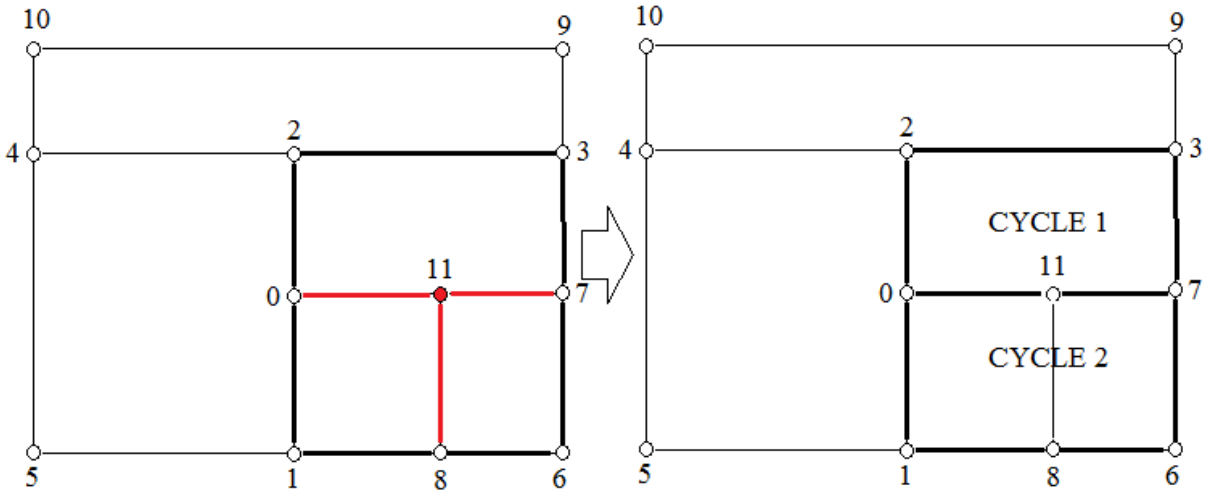
Figure 5.8: Composite Cycle Type 3



Then, the program will rerun CycleType function to analyze the two new cycles and find that the cycle 2→4→5→1→0→2 does not fall under any of the three criteria and thus, it is labeled as simple. However, before storing it in the list of simple cycles, the function IsCycleNew, implemented in lines 570-586 checks whether this cycle has been already found in a previous step. If the cycle is not listed in the simple cycles list, it is added in the list and we move to the next cycle, otherwise we just skip it.

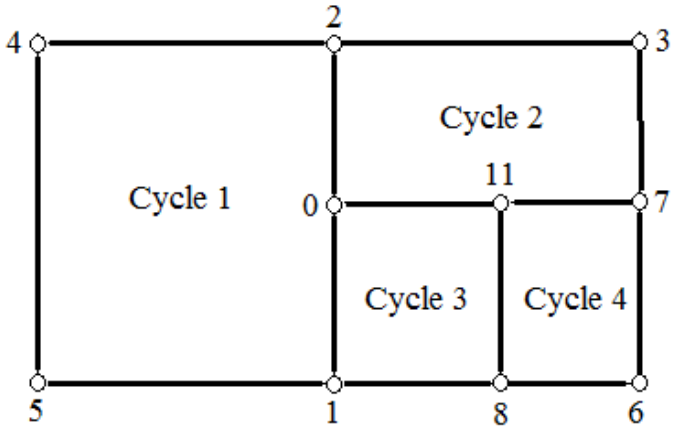
The remaining cylce: 8→6→7→3→2→0→1→8 will be analyzed then. For this cycle, the point "11", which is not a vertex of the cycle, is connected to three vertices of the cycle and thus, the second criterion is activated. As a result, the program divides this cycle into two new ones as it is demonstrated in Figure 5.9.

Figure 5.9: Composite Cycle Division



Then the program will rerun CycleType function and find that Cycle 2 is also composite and divide it further into two new cycles. Finally, it will come up with four simple cycles obtained from one composite cycle 8→6→7→3→2→4→5→1→8. Figure 5.10 depicts the identified simple cycles.

Figure 5.10: Simple Cycles Found



The program stops running when the number of identified simple cycles reaches the expected number, which was calculated by using Euler's formula for planar graphs. Euler's formula states that the number of faces of a graph could be calculated by the equation:

$$V - E + F = 2 \tag{5.1}$$

Where: V – number of vertices; E – number of edges, and F – number of faces.

By rearranging the equation (5.1) and subtracting one from the number of faces, because it also considers the outer face of the graph, the expected number of simple cycles is estimated. This procedure is provided in lines 100-101 of the script.

Finally, when the number of simple cycles reaches the expected one, the program notifies the user and displays the simple cycles found.

The next stage is to add knot intervals for all edges of a T-mesh. To perform this task the following approach was implemented:

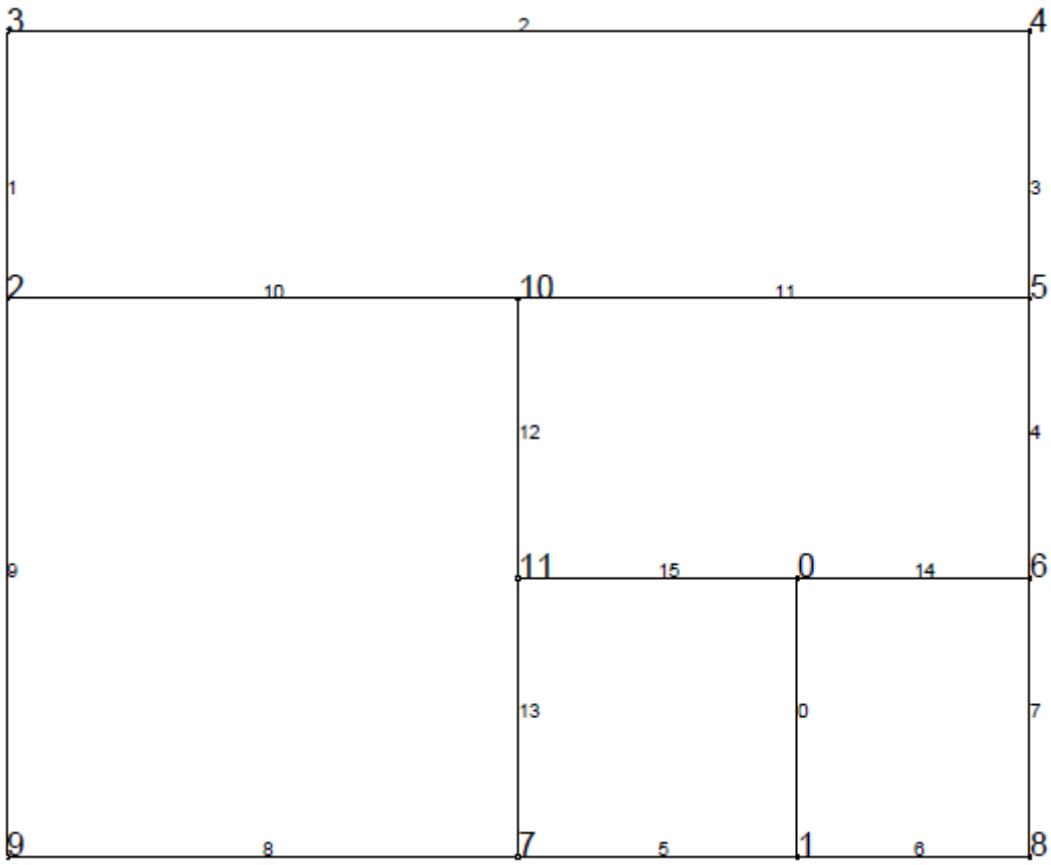
Step 1: Identify the starting cycle and knot origin

Step 2: Divide cycle edges into four categories and assign knot intervals

Step 3: Move to the neighboring cycle and repeat Step 2 till knot intervals are assigned to all edges of the T-mesh.

To describe the above steps, a T-mesh illustrated in Figure 5.11 will be used.

Figure 5.11: Simple T-mesh Example 2



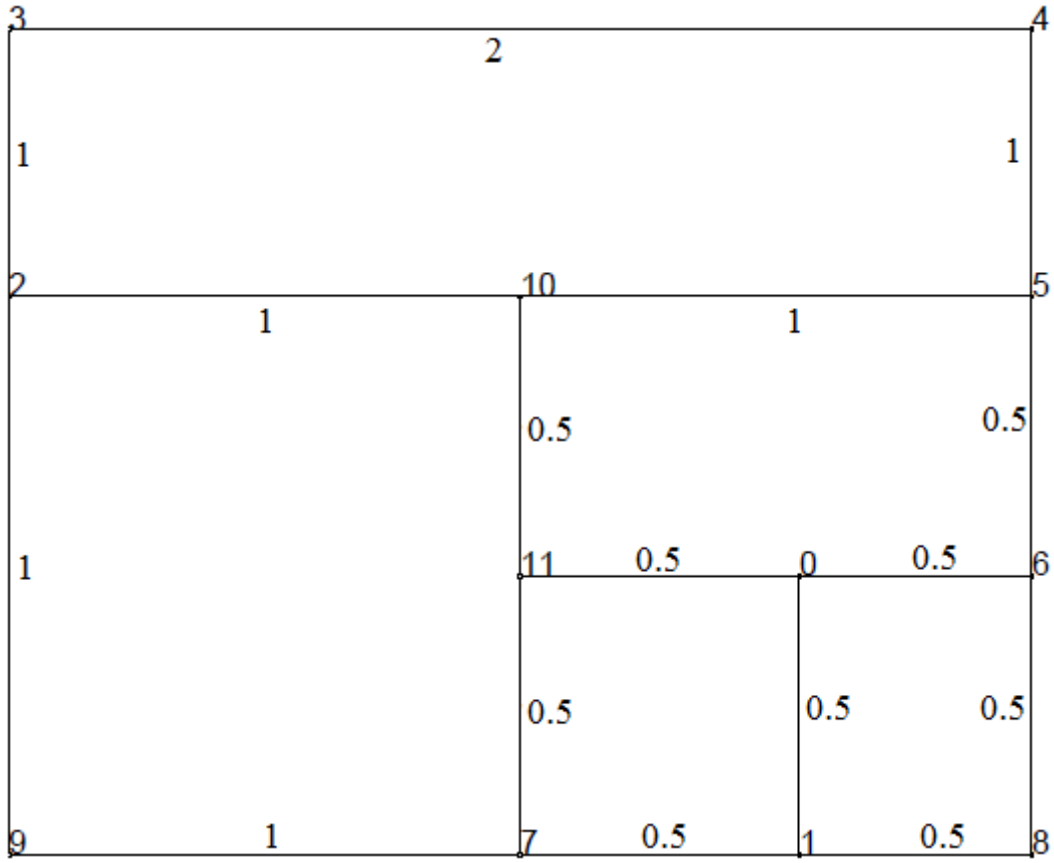
To perform the first step, the program needs to identify an exterior face, i.e., a simple cycle on the boundary of the T-mesh. We do this by calculating the occurrence of each edge in the list of simple cycles. For example, if we want to find an exterior face at the boundary, we need to find the cycle that has at least two exterior edges, i.e., edges that occur only once, that meet to a point with valence 2. At this stage, the program will identify 9→2→10→11→7→9 as a cycle of that kind, as edges 8 and 9 are present only in this cycle and meet at a corner point., Similarly, 8→1→0→6→8 can be also identified due to edges 6 and 7. Then, the program chooses one of them and makes this cycle as the starting one. After that, the program will choose point “9” or “8” as the origin for the

parameter domain depending on what cycle the program selects as the starting one.

Step 2: Let's assume that the starting cycle was selected to be 9→2→10→11→7→9 and the origin was set to point "9". So, the edges of the cycle will be divided into four groups: edge "9" will be marked as "west", edges "12" and "13" will be "east", edge "10" will be "north" and edge "8" will be "south". Then the program will assign knot intervals to each edge following the rule that the sum of all "west" knot intervals must be equal to the sum of "east" ones, and sum of all "north" intervals must be equal to the "south" ones. Then the program will move to the Step 3.

Step 3: After assigning all the knot intervals for the first cycle, the program will move to neighboring cycle, which is 7→11→0→1→7 because it shares edge "13" with the first cycle. Edge "13" was set to be "east" for the first cycle, but for the second cycle it will be marked as "west". Edge "0" will be "east", edge "5" and "15" will be "south" and "north", respectively. Since the program has already assigned a knot interval for edge "13", it will be inherited to the new cycle also. The knot intervals assignment will continue following the approach described above. After the completion of second cycle, the program will move to the next one and repeat the above steps. Finally, when knot intervals are assigned for all edges of the T-mesh, the program comes up with the results depicted in Figure 5.12.

Figure 5.12: Assigned Knot Intervals



Then, by using the knot intervals, we can identify knot coordinates for each control point. For example, point “9” will have (0;0) coordinates, since it was set to be origin, point “11” will have (1;0.5) coordinates and point “5” will have (2;1) knot coordinates.

After that, by using the knot coordinate system, the program will compute knot vectors for all the control points of the T-mesh. Table 5.1 provides the results of this operation.

Table 5.1: Knot vectors

Control Point	u-direction (→)	v-direction (↑)
0	[0,1,1.5,2,2]	[0,0,0.5,1,2]
1	[0,1,1.5,2,2]	[0,0,0,0.5,1]
2	[0,0,0,1,2]	[0,0,1,2,2]
3	[0,0,0,2,2]	[0,1,2,2,2]
4	[0,0,2,2,2]	[0.5,1,2,2,2]
5	[0,1,2,2,2]	[0,0.5,1,2,2]
6	[1,1.5,2,2,2]	[0,0,0.5,1,2]
7	[0,0,1,1.5,2]	[0,0,0,0.5,1]
8	[1,1.5,2,2,2]	[0,0,0,0.5,1]
9	[0,0,0,1,1.5]	[0,0,0,1,2]
10	[0,0,1,2,2]	[0,0.5,1,2,2]
11	[0,0,1,1.5,2]	[0,0,0.5,1,2]

Finally, when the knot vectors are identified for all the control points, the T-spline surface is defined by:

$$P(s,t) = \frac{\sum_{i=1}^{n} w_i P_i B_i(s,t)}{\sum_{i=1}^{n} w_i B_i(s,t)} \tag{5.2}$$

Where: $P_i = (x_i; y_i; z_i;)$ is a control point in cartesian coordinates with a weight of w_i and $B_i(s,t)$ is the blending function given by:

$$B_i(s,t) = N[s_{i0}, s_{i1}, s_{i2}, s_{i3}, s_{i4}](s)N[t_{i0}, t_{i1}, t_{i2}, t_{i3}, t_{i4}](t) \quad (5.3)$$

Which is being calculated by using knot vectors provided in the Table 5.1.

To sum up, T-Spline surface defining process could be divided into three main steps, which are:

1. Add Control Points	• Split Lines • Identify Cycle Edges • Add Control Points
2. Find Simple Cycles	• Create Adjacency Matrix • Run DFS Algorithm • Decompose Composite Cycles
3. Define Blending Functions	• Assign Knot Values • Set Knot Coordinates • Define Knot Intervals

An example of T-spline surface generated for the parameter domain depicted in Figure 5.13 (a) with the physical domain illustrated in Figure 5.13 (b) is provided in Figure 5.14.

Figure 5.13 (a): Parameter Domain

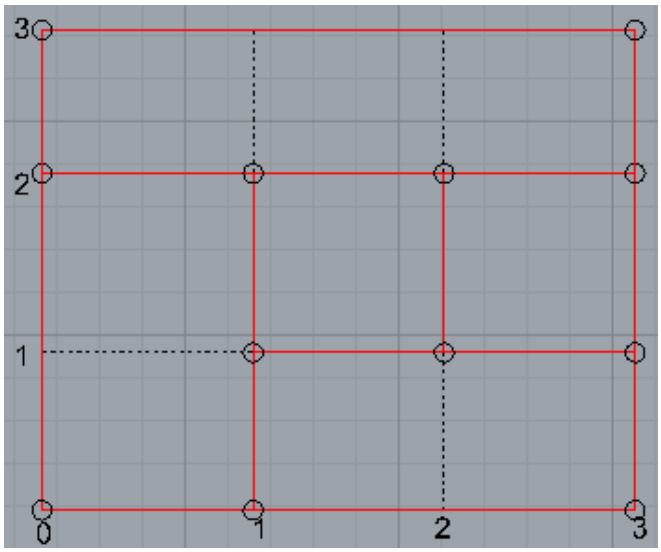


Figure 5.13 (b): Physical Domain

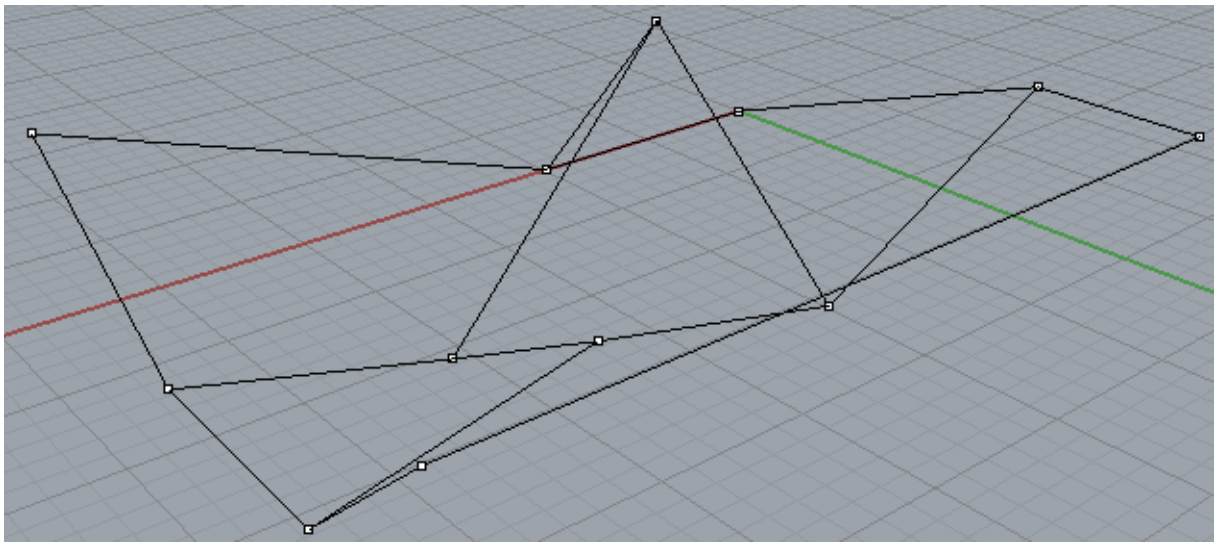
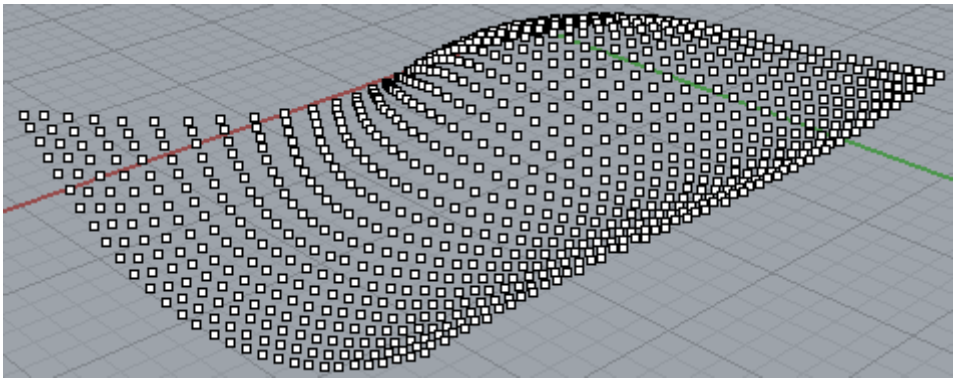


Figure 5.14: T-spline Surface



The following report section demonstrates how the developed script is applied to two different T-mesh examples to find simple cycles.

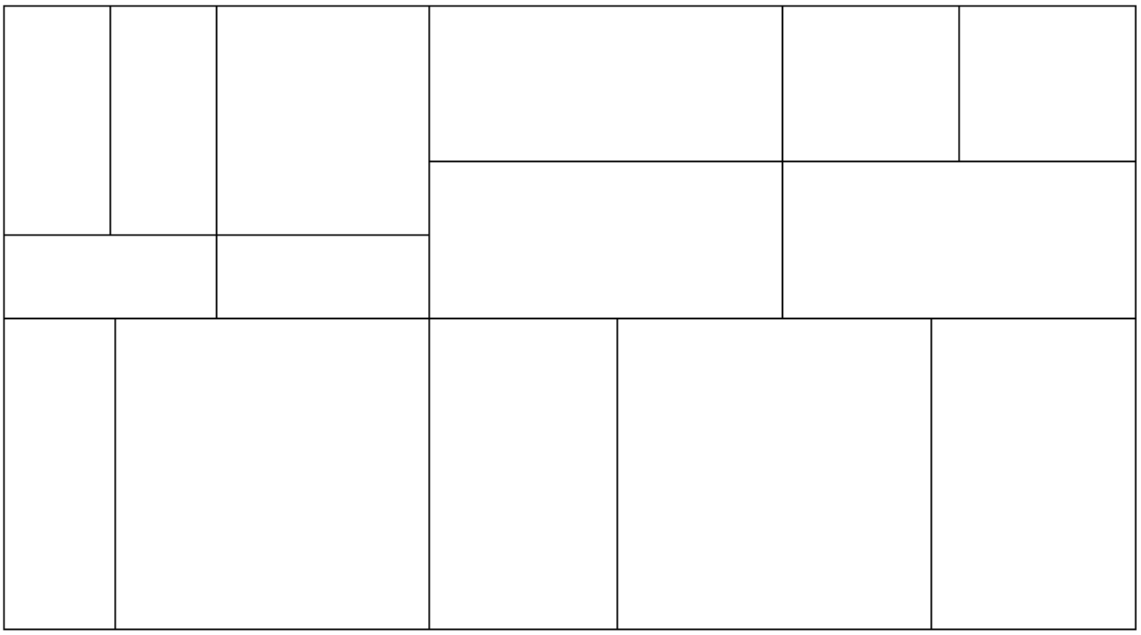
5.2 Rhino Script execution

This section of the report describes how the developed program is applied to two different T-mesh examples and contains the obtained results.

5.2.1 T-mesh Example 1

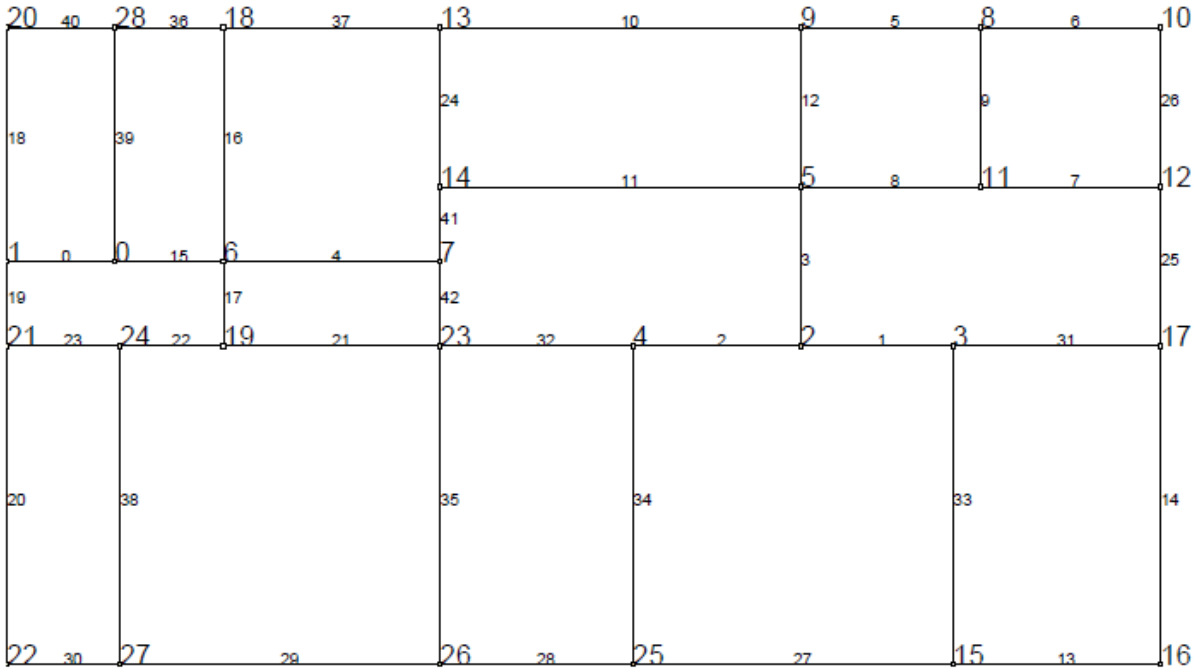
Figure 5.15 illustrates the first T-mesh example which was used to test the operation of the developed script. As it could be manually calculated from the given T-mesh example, it consists of 29 control points and 43 edges which form 15 simple cycles.

Figure 5.15: T-mesh Example 1



When the script starts operating, firstly it puts all the control points on the mesh and prints the numbers of the control points and edges. Figure 5.16 represents the resulting graph.

Figure 5.16: Control Points added



From Figure 5.16 it can be observed that all 29 control points were identified and added to the mesh (points 0-28) and all 43 edges were also identified and numbered (edges 0-42). So, the script works well up to this stage. Then, the program generates and provides the adjacency matrix illustrated in Figure 5.17.

Figure 5.17: Adjacency Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
5	0	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
19	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
20	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
21	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
23	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
25	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
28	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

For example and according to the depicted adjacency matrix, point “0” is connected to points “1”, “6” and “28” which is the case for the given graph. Also, the adjacency matrix shows that point “5” has a connection to the points “2”, “9”, “11” and “14” which is also true for the graph provided. By thoroughly checking all cases we can verify the correctness of the generated adjacency matrix.

After the generation of the adjacency matrix, the program runs the DFS function and finds the basic cycles of the graph. The results of the DFS function are provided in the Table 5.2.

Table 5.2: DFS function Results

Cycle found	Edge #
28→20→1→0→28	4
6→18→28→20→1→0→6	6
17→16→15→3→17	4
9→8→10→12→17→16→15→3→2→5→9	10
13→9→8→10→12→17→16→15→3→2→5→14→13	12
13→9→8→10→12→17→16→15→3→2→5→14→7→6→18→13	15
11→8→10→12→17→16→15→3→2→5→11	10
11→8→10→12→11	4
4→25→15→3→2→4	5
23→4→25→15→3→2→5→14→7→23	9
19→23→4→25→15→3→2→5→14→7→6→19	11
21→24→19→23→4→25→15→3→2→5→14→7→6→18→28→ →2→0→1→21	17
27→22→21→24→27	4
26→27→22→21→24→19→23→26	7
26→27→22→21→24→19→23→4→25→26	9

All the identified basic cycles can be manually checked in our graph. Moreover, it can be observed that each control point is present in at least one identified basic cycle, which indicates that the DFS has covered the whole graph. So, the program moves to the next stage and generates the first set of simple cycles. It simply takes

all the basic cycles with up to five vertices, since such cycles could not be composite. Table 5.3 provides the resulting set of simple cycles.

Table 5.3: Simple Cycles found by DFS

Simple Cycle #	Vertices of the cycle
Cycle 1:	28→20→1→0→28
Cycle 2:	17→16→15→3→17
Cycle 3:	11→8→10→12→11
Cycle 4:	4→25→15→3→2→4
Cycle 5:	27→22→21→24→27

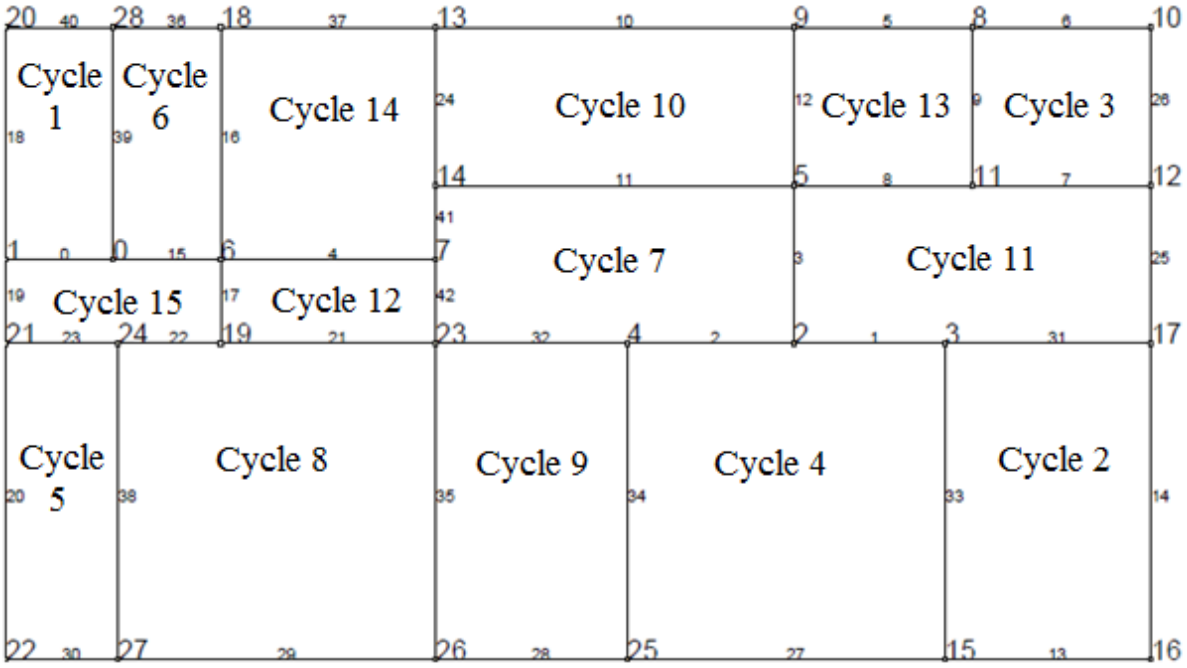
As it can be seen from the Table 5.3, the DFS function has found five simple cycles of the graph. Then, the program calculates the expected number of the simple cycles by Euler's formula described earlier and finds that there should be 15 simple cycles. Since the number of found simple cycles is less than the expected one, the program runs CycleType function to find the remaining cycles. Finally, the script stops operating notifying the user that all 15 simple cycles were found in four iterations of the CycleType function and provides the cycles. The final output of the program is included in Table 5.4.

Table 5.4: Script output

Simple Cycle #	Vertices of the cycle
Cycle 1:	28→20→1→0→28
Cycle 2:	17→16→15→3→17
Cycle 3:	11→8→10→12→11
Cycle 4:	4→25→15→3→2→4
Cycle 5:	27→22→21→24→27
Cycle 6:	6→18→28→0→6
Cycle 7:	23→4→2→5→14→7→23
Cycle 8:	26→27→24→19→23→26
Cycle 9:	26→23→4→25→26
Cycle 10:	13→9→5→14→13
Cycle 11:	11→12→17→3→2→5→11
Cycle 12:	19→23→7→6→19
Cycle 13:	9→8→11→5→9
Cycle 14:	13→14→7→6→18→13
Cycle 15:	21→24→19→6→0→1→21

Figure 5.18 depicts all the simple cycles found in the T-mesh example.

Figure 5.18: Simple Cycles

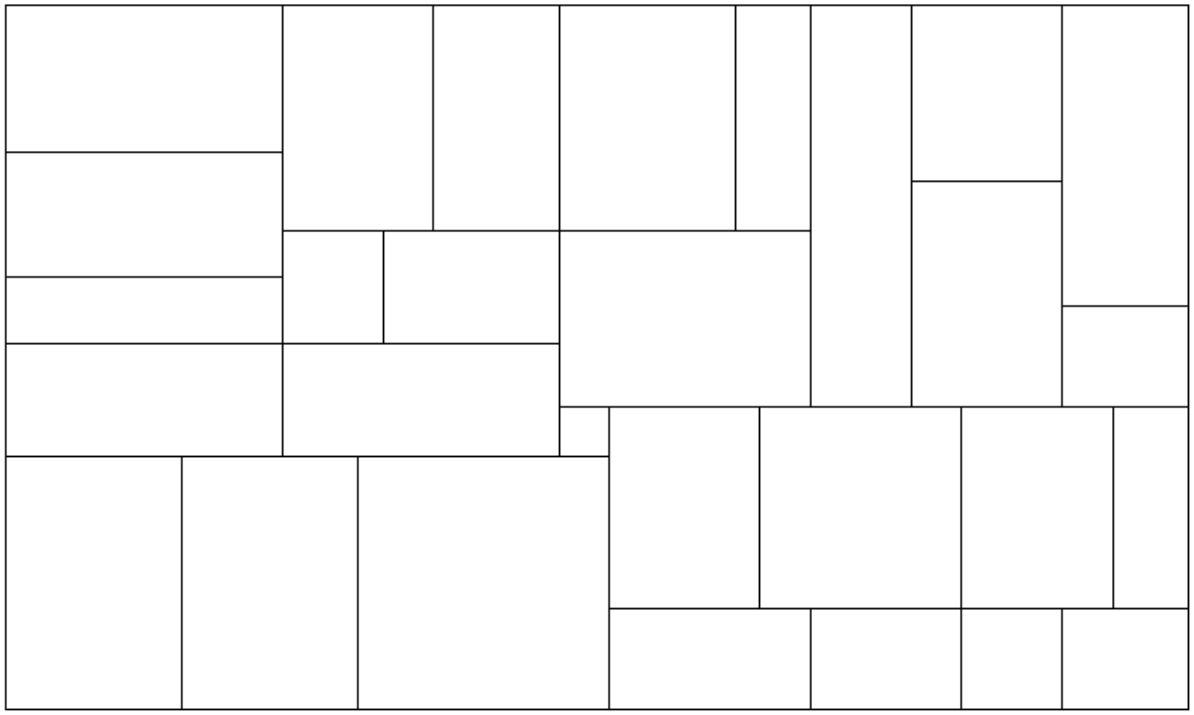


The program was run on a Lenovo laptop with Intel Core™ i3-4030U CPU @ 1.90 GHz (4 CPUs), 4 GB RAM and NVIDIA GeForce 840M graphics processor. It took just two seconds to identify the topological structure.

5.2.2 T-Mesh Example 2

Figure 5.19 illustrates the second T-mesh example which was used to test the operation of the developed script.

Figure 5.19: T-mesh Example 2



In this case, the program identified 57 control points and DFS function found 29 different cycles (See Appendix B), but only 11 of them were simple ones. Figure 5.20 illustrates the T-mesh with added control points, while Table 5.5 provides the simple cycles found by DFS function.

Figure 5.20: T-mesh with Control Points

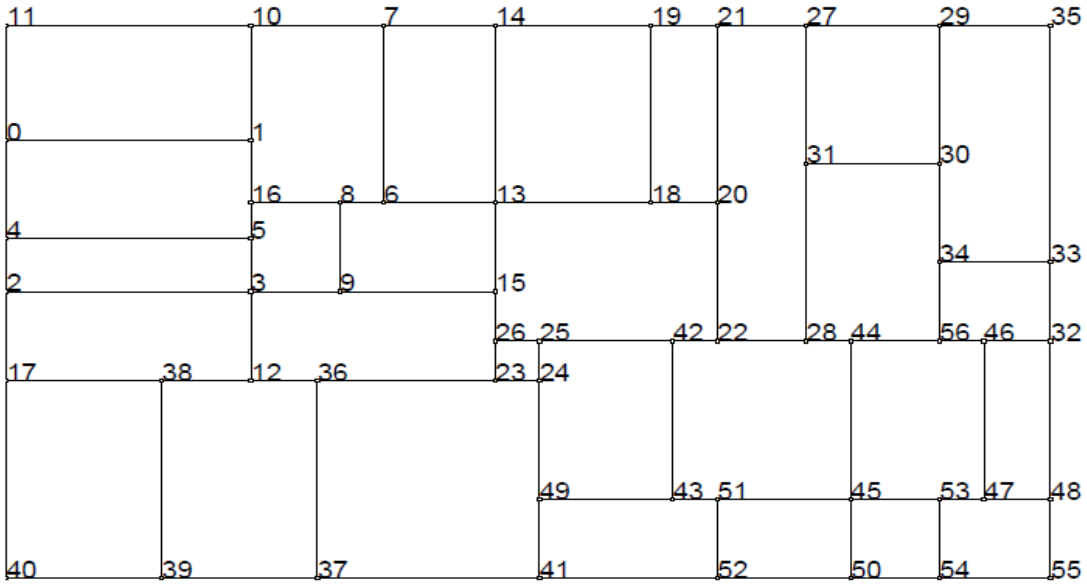


Table 5.5: DFS Function Simple Cycles Results

Simple Cycle #	Vertices of the cycle
Cycle 1:	5→4→2→3→5
Cycle 2:	12→38→17→2→3→12
Cycle 3:	18→19→14→13→18
Cycle 4:	21→20→18→19→21
Cycle 5:	31→30→29→27→31
Cycle 6:	48→47→46→32→48
Cycle 7:	53→54→55→48→47→53
Cycle 8:	46→32→33→34→56→46
Cycle 9:	51→45→50→52→51
Cycle 10:	26→25→24→23→26
Cycle 11:	11→10→1→0→11

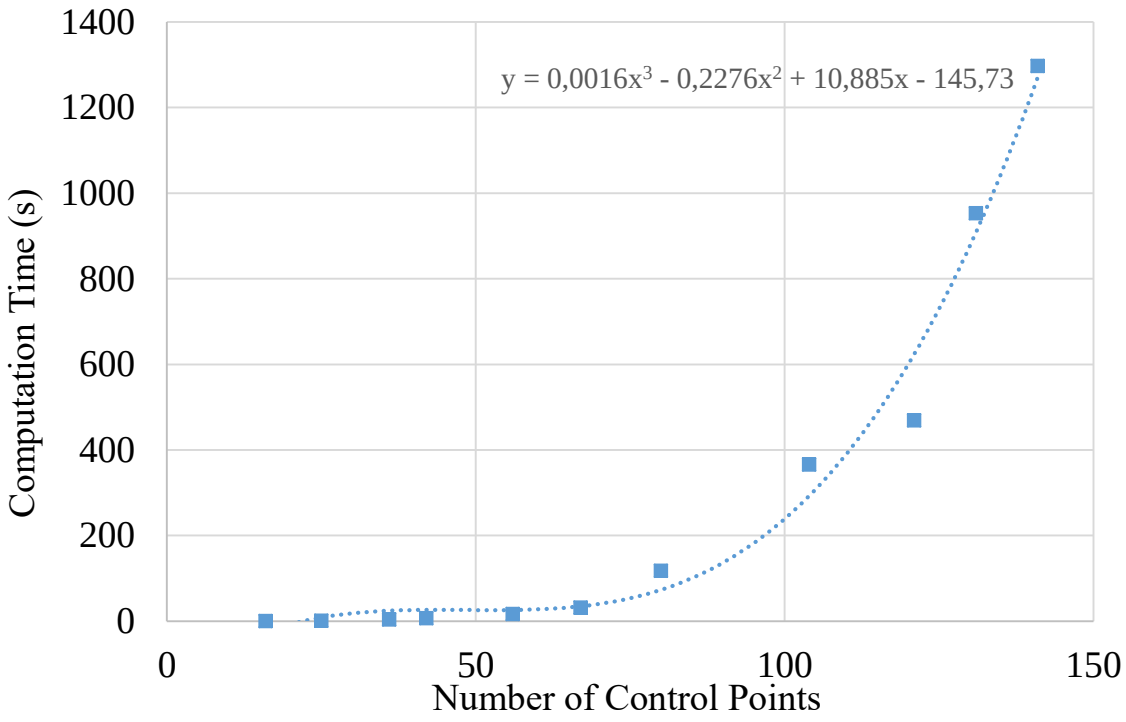
The expected number of simple cycles was calculated to be 29. Thus, the operation of the program continued and CycleType function was run. If took 20 seconds for the program to compute five iterations of the CycleType function and find all 29 simple cycles of the graph. Table 5.6 represents the obtained results.

Table 5.6: All Simple Cycles of the Graph

Simple Cycles	Simple Cycles
1) 5→4→2→3→5	16) 16→8→6→7→10→1→16
2) 12→38→17→2→3→12	17) 15→26→23→36→12→3→9→15
3) 18→19→14→13→18	18) 13→15→9→8→6→13
4) 21→20→18→19→21	19) 22→28→31→27→21→20→22
5) 31→30→29→27→31	20) 42→22→20→18→13→15→26→25→42
6) 48→47→46→32→48	21) 44→45→51→43→42→22→28→44
7) 53→54→55→48→47→53	22) 35→33→34→30→29→35
8) 46→32→33→34→56→46	23) 4→5→16→1→0→4
9) 51→45→50→52→51	24) 49→43→42→25→24→49
10) 26→25→24→23→26	25) 37→41→49→24→23→36→37
11) 11→10→1→0→11	26) 39→37→36→12→38→39
12) 16→5→3→9→8→16	27) 53→47→46→56→44→45→53
13) 14→13→6→7→14	28) 34→56→44→28→31→30→34
14) 40→39→38→17→40	29) 54→53→45→50→54
15) 51→52→41→49→43→51	

Script computation time depends on the complexity of the graph, which is affected by the number of control points and the connectivity between those points. To identify the relation between execution time and number of control points, the worst scenario with the highest connectivity of each control point was analyzed. Figure 5.21 illustrates the obtained results.

Figure 5.21: Control Points vs. Computation Time



The obtained graph has illustrated that time required to identify all the simple cycles of a graph could be approximated by using third degree polynomial.

Chapter 6 – Conclusion

NURBS geometry representation standard in CAD software currently used in Isogeometric Analysis process was analyzed in the report. Also, T-spline surface implementation advantages were identified and the way of defining T-mesh parameters was developed. Moreover, Rhinoscript programming language was used to introduce a T-spline surface in Rhino 5 3D modeler. The developed script is the first step of T-splines implementation in Rhino, its functionality could be enhanced in future work by introducing knot vector identification function described in the Chapter 5. A function that will convert NURBS objects to T-splines and delete all the superfluous control points could also be developed in future work.

Bibliography

[1] Nguyen, V.,P., Bordas, S., and Rabczuk, T., 2015. “Isogeometric Analysis: An Overview and Computer Implementation Aspects”, J. Mathematics and Computers in Simulation, 117, pp. 89-116.

[2] Hughes, T., and Evans, J., 2010. "Isogeometric Analysis", ICES REPORT 10-18, The Institute for Computational Engineering and Sciences, The University of Texas at Austin.

[3] Bazilevs, Y., Calo, V., Cottrell, J., Evans, J., Hughes, T., Lipton, S., Scott, M., and Sederberg, T., 2010. “Isogeometric analysis using T-splines”, J. Computer Methods in Applied Mechanics and Engineering, 199 (5), pp. 229-263.

[4] Hughes, T., Cottrell, J., and Bazilevs, Y., 2005. “Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement”, J. Computer Methods in Applied Mechanics and Engineering, 194, pp. 4135-4195.

[5] Farin, G., 2002. “Curves and Surfaces for CAGD, A Practical Guide”, 5th ed., Morgan Kaufamann.

[6] Joy, K., n.d. “The Geometric Definition of the B-Spline Curve”, Geometric Modeling Lecture notes, University of California.

[7] Zhang, H., and Feng, J., 2006. “B-Spline Interpolation and Approximation”, Digital Geometry Processing course lecture notes, Zhejiang University.

[8] Leal, N., Leal, E., and Branch, J., W., n.d. “Simple Method for Constructing NURBS Surfaces from Unorganized Points”. National University of Colombia.

[9] Rhinoceros, n.d. “What are NURBS” Available at: https://www.rhino3d.com/nurbs [Accessed 2 Dec. 2016].

[10] Piegl, L., 1991. “On NURBS: A Survey”, University of South Florida.

[11] Scott, M., A., 2011. “T-splines as a Design-Through-Analysis Technology”, Ph.D. Thesis, University of Texas at Austin.

[12] Sederberg, T., 2011. “Computer Aided Geometric Design” course notes, Brigham Young University.

[13] Dooren, P., V., 2009. “Graph Theory and Applications” lecture notes. Catholic University of Louvain, Belgium.

[14] Harjy, T., 2011. ‘Graph Theory” lecture notes. University of Turku, Finland.

Appendices

Appendix A

```
Option Explicit
Call Find_Cycles
Sub Find_Cycles()
	Dim i,N,PNTS,pts
	ReDim PNTS(0)
	N = 0
	Const CurveConst = 4
	Dim curve : curve = Rhino.ObjectsByType(CurveConst)
	If IsNull(curve) Then Rhino.Print("Nothing selected")
	If IsArray(curve) Then
		Dim nEdges:nEdges = Ubound(curve)
		For i=0 To Ubound(curve)
			ReDim Preserve PNTS(N)
			PNTS(N) = Rhino.CurveStartPoint(curve(i))
			N = N + 1
			ReDim Preserve PNTS(N)
			PNTS(N) = Rhino.CurveEndPoint(curve(i))
			N = N + 1
		Next
		PNTS = Rhino.CullDuplicatePoints(PNTS)
		pts = Rhino.AddPoints(PNTS)
		'Identify which line connects which Points
		For i=0 To ubound(PNTS)
			Rhino.addText i, PNTS(i), .5
		Next
		Dim j,dist
		Dim e:e = 1e-10
		N = Ubound(PNTS)
		ReDim edge2vertices(nEdges,1)
		ReDim vertices2edge(N,N)
		ReDim index(nEdges)
		For i=0 To nEdges
			index(i) = 0
		Next
		For i = 0 To Ubound(PNTS)
			For j=0 To nEdges
				dist = Rhino.Min(array(Rhino.Distance(PNTS(i),
			Rhino.CurveStartPoint(curve(j))), Rhino.Distance(PNTS(i),
			Rhino.CurveEndPoint(curve(j)))))
				If dist < e Then
					edge2vertices(j, index(j)) = i
					index(j) = 1
				End If
			Next
		Next
		'Create Matrix
```

```
47              ReDim Matrix(N,N)
48              For i=0 To N
49                      For j=0 To N
50                              Matrix(i, j) = 0
51                              vertices2edge(i, j) = -1
52                      Next
53              Next
54              For j=0 To nEdges
55                      Rhino.AddText j, Rhino.CurveMidPoint(curve(j)), 0.3
56                      vertices2edge(edge2vertices(j, 0), edge2vertices(j, 1)) = j
57                      vertices2edge(edge2vertices(j, 1), edge2vertices(j, 0)) = j
58              Next
59              For i=0 To nEdges
60                      Matrix(edge2vertices(i, 0), edge2vertices(i, 1)) = 1
61                      Matrix(edge2vertices(i, 1), edge2vertices(i, 0)) = 1
62              Next
63              'Print Matrix
64              Dim str
65              str = ""
66              For i=0 To N
67                      For j=0 To N
68                              str = str & Matrix(i, j)
69                      Next
70                      Rhino.Print str
71                      str = ""
72              Next
73              'Run DFS to find basic cyles
74              Dim basic_cycles:basic_cycles = DFS(Matrix, N, vertices2edge, nEdges)
75              ReDim simple_cycles(0)
76
77              'Find simple cycles in basic cycles
78              'All basic cycles with up to 5 edges are simple
79              Dim sc
80              sc = 0
81              For i=0 To ubound(basic_cycles)
82                      If basic_cycles(i)(0) < 6 Then
83                              If ubound(simple_cycles) < sc Then
84                                      ReDim Preserve simple_cycles(sc)
85                              End If
86                              simple_cycles(sc) = basic_cycles(i)
87                              sc = sc + 1
88                      End If
89              Next
90              'create sorted version of simple cycles
91              ReDim sorted_s_c(0)
92              Dim sort
93              sort = 0
94              For sc=0 To Ubound(simple_cycles)
95                      ReDim Preserve sorted_s_c(sort)
96                      sorted_s_c(sort) = rhino.SortNumbers(simple_cycles(sc)(2))
```

```
97                  sort = sort + 1
98              Next
99              'find the expected number of simple cycles
100             Dim Expected_number,difference
101             Expected_number = 2 - (N + 1) + (nEdges + 1) - 1
102             If Expected_number - (ubound(simple_cycles) + 1) = 0 Then
103                 Rhino.Print "All simple cycles are found"
104             End If
105             'then identify whether the cycle is composite or simple
106             ReDim composite_cycles(0)
107             Dim cc
108             cc = 0
109             ReDim new_cycle(0)
110             Dim nc
111             nc = 0
112             For i=0 To ubound(basic_cycles)
113                 cycle_type i, basic_cycles, vertices2edge, composite_cycles, cc,
114             simple_cycles, sc, new_cycle, nc, PNTS, sorted_s_c, sort
115             Next
116             Dim iteration
117             For iteration=1 To 10
118                 Erase basic_cycles
119                 For i=0 To Ubound(new_cycle)
120                     ReDim Preserve basic_cycles(i)
121                     basic_cycles(i) = new_cycle(i)
122                 Next
123                 Erase new_cycle
124                 nc = 0
125                 For i=0 To ubound(basic_cycles)
126                     'If basic_cycles(i)(0) > 5 Then  'number of vertices is at least 6
127                     cycle_type i, basic_cycles, vertices2edge, composite_cycles, cc,
128                 simple_cycles, sc, new_cycle, nc, PNTS, sorted_s_c, sort
129                     'End If
130                 Next
131                 Difference = Expected_number - (ubound(simple_cycles) + 1)
132                 If difference = 0 Then
133                     Exit For
134                 End If
135             Next
136             For sc=0 To ubound(simple_cycles)
137                 Rhino.print "Simple Cycle: " & Cycle2String(simple_cycles(sc)(2))
138             Next
139             If difference = 0 Then
140                 Rhino.Print "All " & Expected_number & " Simple Cycles found in " &
141             iteration & " iterations"
142             Else
143                 Rhino.Print "Cycles not found: " & Difference
144                 Rhino.Print "Redraw some of your lines and rerun the script"
145             End If
146             ""
```

```
147                 'edgInCycle=simple cycles represented in terms of edges, not points
148                 ReDim edgInCycle(0)
149                 Dim ed
150                 ed = 0
151                 For sc=0 To ubound(simple_cycles)
152                         EdgesInCycle sc, simple_cycles, edgInCycle, vertices2edge, ed
153                 Next
154                 'find the number of cycles in which each edge present
155                 ReDim occ(0)
156                 Dim num
157                 num = 0
158                 For i=0 To nEdges
159                         For ed=0 To ubound(edgInCycle)
160                                 For j=0 To ubound(edgInCycle(ed))
161                                         If i = edgInCycle(ed)(j) Then
162                                                 num = num + 1
163                                         End If
164                                 Next
165                         Next
166                         ReDim Preserve occ(i)
167                         occ(i) = num
168                         num = 0
169                 Next
170                 'find starting cycle (any of the points which present only in one cycle)
171                 ReDim Cycles(0)
172                 Dim order
173                 order = 0
174                 ReDim Preserve Cycles(order)
175                 Cycles(order) = StartCycle(edgInCycle, occ, ed, i, j, simple_cycles)
176                 Rhino.Print "Starting Cycle: " & Cycle2String(cycles(0)(0))
177
178         End If
179 End Sub
180
181 Function DFS(A, N, v2e, N1)
182         Dim i
183         ReDim cycles(0)
184         ReDim v_prev(N)
185         ReDim v_color(N)
186         For i=0 To N
187                 v_color(i) = 0 '0:white, 1:gray, 2:black
188                 v_prev(i) = -1 '-1: no parent
189         Next
190         For i=0 To N
191                 If (v_color(i) = 0) Then
192                         DFS_visit i, A, N, v_color, v_prev, cycles, v2e, N1
193                 End If
194         Next
195         ReDim Preserve cycles(ubound(cycles)-1)
196         DFS = cycles
```

```
End Function

Function DFS_Visit(u, A, N, C, P, cycles, v2e, N1)
	C(u) = 1
	Dim i,data,index,count,str:str = ""
	Dim con:con = GetConnected(u, A, N)
	For i=0 To ubound(con)
		If C(con(i)) = 1 And P(u) <> con(i) Then
			Rhino.print "Cycle found"
			index = ubound(cycles) + 1
			data = ExtractCycle(con(i), u, P, v2e, N1)
			count = CountCycleEdges(data(0))
			Rhino.print data(1) & " Edge Count: " & count
			ReDim Preserve cycles(index)
			cycles(index - 1) = array(count, data(0), data(2), data(3))
		End If
		If (C(con(i)) = 0) Then
			P(con(i)) = u
			DFS_visit con(i), A, N, C, P, cycles, v2e, N1
		End If
	Next
	C(u) = 2
End Function

Function GetConnected(u, A, N)
	Dim i,j
	ReDim C(1)
	j = 0
	For i=0 To N
		If (A(u, i) = 1) Then
			If j > ubound(C) Then
				ReDim Preserve C(j)
			End If
			C(j) = i
			j = j + 1
		End If
	Next
	GetConnected = C
End Function

Function ExtractCycle(a, c, Previous, v2e, N1)
	Dim str,i:str = ""
	ReDim vec(N1)
	ReDim pvec(0)
	For i=0 To N1
		vec(i) = 0
	Next
	i = 0
	Dim b:b = c
	While Previous(b) <> a
```

```
247                 vec(v2e(b, Previous(b))) = 1
248                 str = str & b & "->"
249                 If (ubound(pvec) < i) Then
250                         ReDim Preserve pvec(i)
251                 End If
252                 pvec(i) = b
253                 i = i + 1
254                 b = Previous(b)
255
256         Wend
257         ReDim Preserve pvec(i+1)
258         pvec(i) = b
259         pvec(i + 1) = Previous(b)
260         str = str & b & "->" & Previous(b) & "->" & c
261         vec(v2e(b, Previous(b))) = 1
262         vec(v2e(Previous(b), c)) = 1
263         Dim inc_v,j
264         inc_v = ""
265         For j=0 To 0
266                 For i=0 To N1
267                         inc_v = inc_v & vec(i)
268                 Next
269         Next
270         ExtractCycle = array(vec, str, pvec, inc_v)
271 End Function
272
273 Function CountCycleEdges(c)
274         Dim i,size:size = 0
275         For i = 0 To ubound(c)
276                 size = size + c(i)
277         Next
278         CountCycleEdges = size
279 End Function
280
281 Function StartCycle(edgInCycle, occ, ed, i, j, simple_cycles)
282         For ed=0 To ubound(edgInCycle)
283                 For i=0 To ubound(edgInCycle(ed))
284                         For j=0 To ubound(edgInCycle(ed))
285                                 If i <> j Then
286                                         If occ(edgInCycle(ed)(i)) = 1 Then
287                                                 If occ(edgInCycle(ed)(j)) = 1 Then
288                                                         StartCycle = array(simple_cycles(ed)(2),
289                                                 edgInCycle(ed))
290                                                         Exit Function
291                                                 End If
292                                         End If
293                                 End If
294                         Next
295                 Next
296         Next
```

```
297 End Function
298
299 Function EdgesInCycle(sc, simple_cycles, edgInCycle, vertices2edge, ed)
300         Dim j
301
302         Dim ed_in
303         ed_in = 0
304         ReDim edges_in(0)
305         For j=0 To ubound(simple_cycles(sc)(2)) - 1
306                 ReDim Preserve edges_in(ed_in)
307                 edges_in(ed_in) = vertices2edge(simple_cycles(sc)(2)(j),
308                                 simple_cycles(sc)(2)(j+1))
309                 ed_in = ed_in + 1
310         Next
311         ReDim Preserve edges_in(ed_in)
312         edges_in(ed_in) = vertices2edge(simple_cycles(sc)(2)(ubound(simple_cycles(sc)(2))),
313                   simple_cycles(sc)(2)(0))
314         ReDim Preserve edgInCycle(ed)
315         edgInCycle(ed) = edges_in
316         ed = ed + 1
317 End Function
318
319 Function cycle_type(i, b_c, vertices2edge, composite_cycles, cc, simple_cycles, sc,
320 new_cycle, nc, PNTS, sorted_s_c, sort)
321         Dim j,k,l,m,o,c1,c2,OPS,ops2,boundaries,vvv,ops3,c3,aaa,bbb,ccc,c4,c5,c6
322         vvv = 0
323         ops2 = -1
324         ReDim cycle1(0)
325         c1 = 0
326         ReDim cycle2(0)
327         c2 = 0
328         ReDim cycle3(0)
329         c3 = 0
330         ReDim cycle4(0)
331         c4 = 0
332         ReDim cycle5(0)
333         c5 = 0
334         ReDim cycle6(0)
335         c6 = 0
336         For j=0 To ubound(PNTS) 'find any points that is connected to 3 vertices of a cycle
337                 For k=0 To ubound(b_c(i)(2))
338                         For l=0 To ubound(b_c(i)(2))
339                                 For m=0 To ubound(b_c(i)(2))
340                                         If j <> b_c(i)(2)(k) And j <> b_c(i)(2)(l)
341                   And j <> b_c(i)(2)(m) And b_c(i)(2)(k) <> b_c(i)(2)(l)
342                   And b_c(i)(2)(k) <> b_c(i)(2)(m)
343                   And b_c(i)(2)(l) <> b_c(i)(2)(m)
344                   And vertices2edge(j, b_c(i)(2)(k)) > -1
345                   And vertices2edge(j, b_c(i)(2)(l)) > -1
346                   And vertices2edge(j, b_c(i)(2)(m)) > -1 Then
```

```
                ReDim Preserve composite_cycles(cc)
                composite_cycles(cc) = b_c(i)
                cc = cc + 1
                For ops=0 To ubound(b_c(i)(2))
                        If j = b_c(i)(2)(ops) Then
                                If m - ops > 1 Or ops - m > 1 Then
                                If m - ops <> ubound(b_c(i)(2))
And ops - m <> ubound(b_c(i)(2)) Then
                                                ops2 = m
                                End If
                        End If
                        If l - ops > 1 Or ops - l > 1 Then
                                If l - ops <> ubound(b_c(i)(2))
 And ops - l <> ubound(b_c(i)(2)) Then
                                                ops2 = l
                                End If
                        End If
                        If k - ops > 1 Or ops - k > 1 Then
                                If k - ops <> ubound(b_c(i)(2))
And ops - k <> ubound(b_c(i)(2)) Then
                                                ops2 = k
                                End If
                        End If
                        If ops2 > -1 Then
                                boundaries = array(ops, ops2)

 For o=Rhino.Min(boundaries) To
                Rhino.Max(boundaries)
                 ReDim Preserve cycle1(c1)
                 cycle1(c1) = b_c(i)(2)(o)
                 c1 = c1 + 1
                 Next
                 ReDim Preserve new_cycle(nc)
                 new_cycle(nc) = array(0, 0, cycle1)
                 nc = nc + 1

 For o=0 To Rhino.Min(boundaries)  'second cycle
                 ReDim Preserve cycle2(c2)
                 cycle2(c2) = b_c(i)(2)(o)
                 c2 = c2 + 1
 Next
 For o = Rhino.Max(boundaries) To ubound(b_c(i)(2))
                 ReDim Preserve cycle2(c2)
                 cycle2(c2) = b_c(i)(2)(o)
                 c2 = c2 + 1
 Next
 ReDim Preserve new_cycle(nc)
 new_cycle(nc) = array(0, 0, cycle2)
 nc = nc + 1
 boundaries = ""
```

```
397                     Exit Function
398                     End If
399             End If
400                     Next
401                     vvv = 0
402                     For ops=0 To ubound(b_c(i)(2))
403                             If j = b_c(i)(2)(ops) Then
404                                     vvv = 1
405                             End If
406                     Next
407                     If vvv = 0 Then 'Comp TYPE2
408                                     If k < l And k < m And l < m Then
409                                     aaa = k
410                                     bbb = l
411                                     ccc = m
412                             End If
413                             If k < l And k < m And m < l Then
414                                     aaa = k
415                                     bbb = m
416                                     ccc = l
417                             End If
418                             If l < k And l < m And k < m Then
419                                     aaa = l
420                                     bbb = k
421                                     ccc = m
422                             End If
423                             If l < m And l < k And m < k Then
424                                     aaa = l
425                                     bbb = m
426                                     ccc = k
427                             End If
428                             If m < k And m < l And k < l Then
429                                     aaa = m
430                                     bbb = k
431                                     ccc = l
432                             End If
433                             If m < l And m < k And l < k Then
434                                     aaa = m
435                                     bbb = l
436                                     ccc = k
437                             End If
438                             For ops3=0 To aaa
439                                     ReDim Preserve cycle4(c4)
440                                     cycle4(c4) = b_c(i)(2)(ops3)
441                                     c4 = c4 + 1
442                             Next
443                             ReDim Preserve cycle4(c4)
444                             cycle4(c4) = j
445                             c4 = c4 + 1
446                             For ops3=bbb To ubound(b_c(i)(2))
```

```
                                        ReDim Preserve cycle4(c4)
                                        cycle4(c4) = b_c(i)(2)(ops3)
                                        c4 = c4 + 1
                                    Next
                                    ReDim Preserve new_cycle(nc)
                                    new_cycle(nc) = array(0, 0, cycle4)
                                    nc = nc + 1
                                    For ops3=0 To bbb
                                        ReDim Preserve cycle5(c5)
                                        cycle5(c5) = b_c(i)(2)(ops3)
                                        c5 = c5 + 1
                                    Next
                                    ReDim Preserve cycle5(c5)
                                    cycle5(c5) = j
                                    c5 = c5 + 1
                                    For ops3=ccc To ubound(b_c(i)(2))
                                        ReDim Preserve cycle5(c5)
                                        cycle5(c5) = b_c(i)(2)(ops3)
                                        c5 = c5 + 1
                                    Next
                                    ReDim Preserve new_cycle(nc)
                                    new_cycle(nc) = array(0, 0, cycle5)
                                    nc = nc + 1
                                    Exit Function
                                End If
                                vvv = 0
                                End If
                            Next
                        Next
                    Next
                Next
                Dim jj,kk,ll,mm
                For jj=0 To ubound(PNTS)
                    For kk=0 To ubound(b_c(i)(2))
                        For ll=0 To ubound(b_c(i)(2))
                            vvv = 0
                            For ops=0 To ubound(b_c(i)(2))
                                If jj = b_c(i)(2)(ops) Then
                                    vvv = 1
                                End If
                            Next
                            If vvv = 0 Then
                                If jj <> b_c(i)(2)(kk) And jj <> b_c(i)(2)(ll) And kk <> ll
                             And vertices2edge(jj, b_c(i)(2)(kk)) > -1
                             And vertices2edge(jj, b_c(i)(2)(ll)) > -1 Then
                                    ReDim Preserve composite_cycles(cc)
                                    composite_cycles(cc) = b_c(i)
                                    cc = cc + 1
                                    If kk < ll Then
                                        aaa = kk
```

```
497                                                 bbb = ll
498                                             End If
499                                             If ll < kk Then
500                                                 aaa = ll
501                                                 bbb = kk
502                                             End If
503                                             For ops3=0 To aaa 'cycle 1
504                                                 ReDim Preserve cycle3(c3)
505                                                 cycle3(c3) = b_c(i)(2)(ops3)
506                                                 c3 = c3 + 1
507                                             Next
508                                             ReDim Preserve cycle3(c3)
509                                             cycle3(c3) = jj
510                                             c3 = c3 + 1
511                                             For ops3=bbb To ubound(b_c(i)(2))
512                                                 ReDim Preserve cycle3(c3)
513                                                 cycle3(c3) = b_c(i)(2)(ops3)
514                                                 c3 = c3 + 1
515                                             Next
516
517                                             ReDim Preserve new_cycle(nc)
518                                             new_cycle(nc) = array(0, 0, cycle3)
519                                             nc = nc + 1
520                                             'cycle 2
521                                             For ops3=aaa To bbb
522                                                 ReDim Preserve cycle6(c6)
523                                                 cycle6(c6) = b_c(i)(2)(ops3)
524                                                 c6 = c6 + 1
525                                             Next
526                                             ReDim Preserve cycle6(c6)
527                                             cycle6(c6) = jj
528                                             c6 = c6 + 1
529
530                                             ReDim Preserve new_cycle(nc)
531                                             new_cycle(nc) = array(0, 0, cycle6)
532                                             nc = nc + 1
533                                             Exit Function
534                                         End If
535                                     End If
536                                 Next
537                             Next
538                         Next
539                         Dim cycle_sorted
540                         cycle_sorted = Rhino.SortNumbers(b_c(i)(2))
541                         If isCycleNew(cycle_sorted, sorted_s_c) Then
542                             ReDim Preserve sorted_s_c(sort)
543                             sorted_s_c(sort) = cycle_sorted
544                             sort = sort + 1
545                             ReDim Preserve simple_cycles(sc)
546                             simple_cycles(sc) = b_c(i)
```

```
547                     sc = sc + 1
548             End If
549     End Function
550
551     Function Factorial(n)
552             Dim Result
553             Result = 1
554             Do While n > 0
555                     Result = Result * (n)
556                     n = n - 1
557             Loop
558             Factorial = Result
559     End Function
560
561     Function Cycle2String(c)
562             Dim i,str:str = ""
563             For i=0 To ubound(c)
564                     str = str & c(i) & "->"
565             Next
566             str = str & c(0)
567             Cycle2String = str
568     End Function
569
570     Function isCycleNew(n_c, s_c)
571             isCycleNew = True
572             Dim q,w,k
573             k = 0
574             For q=0 To ubound(s_c)
575                     If ubound(n_c) = ubound(s_c(q)) Then
576                             For w=0 To ubound(n_c)
577                                     If n_c(w) = s_c(q)(w) Then
578                                             k = k + 1
579                                     End If
580                                     If k = ubound(n_c) Then
581                                             isCycleNew = False
582                                     End If
583                             Next
584                     End If
585             Next
586     End Function
```

Appendix B

DFS function Results

Cycle 1: 4→2→3→9→8→6→7→10→1→0→4 Edge Count: 10

Cycle 2: 5→4→2→3→5 Edge Count: 4

Cycle 3: 16→5→4→2→3→9→8→6→7→10→1→16 Edge Count: 11

Cycle 4: 16→5→4→2→3→9→8→16 Edge Count: 7

Cycle 5: 12→38→17→2→3→12 Edge Count: 5

Cycle 6: 15→26→25→24→23→36→12→38→17→2→3→9→15 Edge Count: 12

Cycle 7: 13→15→26→25→24→23→36→12→38→17→2→3→9→8→6→13 Edge Count: 15

Cycle 8: 14→13→15→26→25→24→23→36→12→38→17→2→3→9→8→6→7→14 Edge Count: 17

Cycle 9: 18→19→14→13→18 Edge Count: 4

Cycle 10: 21→20→18→19→21 Edge Count: 4

Cycle 11: 31→30→29→27→31 Edge Count: 4

Cycle 12: 22→28→31→30→29→27→21→20→22 Edge Count: 8

Cycle 13: 42→22→28→31→30→29→27→21→20→18→19→14→13→15→26→25→42 Edge Count: 16

Cycle 14:
49→43→42→22→28→31→30→29→27→21→20→18→19→14→13→15→26→25→24→49 Edge Count: 19

Cycle 15:
37→41→49→43→42→22→28→31→30→29→27→21→20→18→19→14→13→15→26→25→24→23→36→37 Edge Count: 23

Cycle 16:
39→37→41→49→43→42→22→28→31→30→29→27→21→20→18→19→14→13→15→26→25→24→23→36→12→38→39 Edge Count: 26

Cycle 17:
40→39→37→41→49→43→42→22→28→31→30→29→27→21→20→18→19→14→13→15→26→25→24→23→36→12→38→17→40 Edge Count: 28

Cycle 18: 44→45→50→52→41→49→43→42→22→28→44 Edge Count: 10

Cycle 19: 34→56→44→45→50→52→41→49→43→42→22→28→31→30→34 Edge Count: 14

Cycle 20: 48→47→46→32→48 Edge Count: 4

Cycle 21: 54→55→48→47→46→32→33→34→56→44→45→50→54 Edge Count: 12

Cycle 22: 53→54→55→48→47→46→32→33→34→56→44→45→53 Edge Count: 12

Cycle 23: 53→54→55→48→47→53 Edge Count: 5

Cycle 24: 46→32→33→34→56→46 Edge Count: 5

Cycle 25: 35→33→34→56→44→45→50→52→41→49→43→42→22→28→31→30→29→35 Edge Count: 17

Cycle 26: 51→45→50→52→41→49→43→51 Edge Count: 7

Cycle 27: 51→45→50→52→51 Edge Count: 4

Cycle 28: 26→25→24→23→26 Edge Count: 4

Cycle 29: 11→10→1→0→11 Edge Count: 4