Projection, Design and Representation of Curves on B-Spline Surfaces

by

Jie Wen Tjung

Thesis Submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Mechanical Engineering

APPROVED:

[Signature]

Dr. A. Myklebust, Chairman

[Signature]        [Signature]

Dr. S. Jayaram       Dr. C. Reinholdtz

January 11, 1993

Blacksburg, Virginia
Projection, Design and Representation of Curves on B-Spline Surfaces

by

Jie Wen Tjung

Dr. A. Myklebust, Chairman

Mechanical Engineering

(ABSTRACT)

In an interactive design process, tools that provide visual feedback on the design model are highly valuable. This research explores a specific topic of graphical visualization and geometric modeling. An approach dealing with projecting and mapping plane curves onto a B-spline surface is presented. The curve is mapped onto the surface using normal projection. All points on the curve are projected onto the surface in the direction of the plane normal. Intersections of projected lines and the surface patches are then found. For an open surface, the curve need not be contained in the interior of the design surface, for if it is mapped out of the surface boundaries, a novel approach has been devised to determine the appropriate edge boundaries of the surface to close the out-of-bounds mapped curve. The issue of mapping curves onto a surface provides a means of attaining a higher level of detailed visualization in a design process, such as the visual feedback on the placement of control surfaces on aircraft wings in an aircraft conceptual design process. The mapped curves can also be treated as trim curves on the surface. This research also proposes an idea for approximating the trim curve with the iso-parametric line segments of the surface to address a probable solution to the many problems facing geometrical surface trimming. The complete curve mapping process has been implemented in the conceptual aircraft design software ACSYNT (AirCraft SYNThesis).
Acknowledgments

Few parents have sacrificed so much to put their children through college as my parents, Chung Yen Chin and Tjiang Geok Eng have. I am forever indebted to them. I also cannot thank enough of the support and love that my brother Jee Chean, sister Ping Ling and brother-in-law koko Arief have given me.

I would like to thank my advisor Dr. Arvid Myklebust for believing in me and providing me with financial assistance in completing my Master's degree. Thanks are extended to Dr. Sankar Jayaram and Dr. Charles Reinholtz, not only for being in my graduate committee but more importantly, for their frequent invaluable advice.

Special thanks to Dr. Wenhsyong Lin, whose explanations and suggestions of ideas have been very valuable, Scott Woyak, Parasuram, David Coe, Steve Fleming and all the CAD lab staff. You have all made my stay in America, especially in Blacksburg, a very informative one.

Last but not least, I would like to thank Bing. If not for her, I would have forgotten the taste of 'real' food for the past couple of months!
Table of Contents

1.0 Introduction ............................................................................................................. 1

2.0 Problem Definition, Research Objectives and Thesis Organization .......... 3
  2.1 Problem Definition .......................................................................................... 3
  2.2 Research Objectives ....................................................................................... 5
  2.3 Thesis Organization ....................................................................................... 6

3.0 Literature Review .................................................................................................. 7

4.0 ACSYNT ................................................................................................................. 11
  4.1 ACSYNT .......................................................................................................... 11
  4.2 ACSYNT B-Spline Module ............................................................................. 12

5.0 Parametric B-Splines ......................................................................................... 14
  5.1 Cubic Parametric Equations ........................................................................... 14
  5.2 B-Splines ........................................................................................................ 16
  5.3 B-Spline Curves ............................................................................................. 17
  5.4 Constraint-Based B-Spline Curve Inversion .................................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.4</td>
<td>define_point.c</td>
<td>124</td>
</tr>
<tr>
<td>B.4.1</td>
<td>define_point</td>
<td>124</td>
</tr>
<tr>
<td>B.4.2</td>
<td>get_patch_picked</td>
<td>125</td>
</tr>
<tr>
<td>B.5</td>
<td>draw_bspline_comp.c</td>
<td>126</td>
</tr>
<tr>
<td>B.5.1</td>
<td>draw_bspline_comp</td>
<td>126</td>
</tr>
<tr>
<td>B.5.2</td>
<td>display_colorful_comp</td>
<td>127</td>
</tr>
<tr>
<td>B.6</td>
<td>draw_curve.c</td>
<td>128</td>
</tr>
<tr>
<td>B.6.1</td>
<td>display_curve</td>
<td>128</td>
</tr>
<tr>
<td>B.6.2</td>
<td>draw_curve</td>
<td>128</td>
</tr>
<tr>
<td>B.6.3</td>
<td>find_knot</td>
<td>129</td>
</tr>
<tr>
<td>B.7</td>
<td>drawMapped_curve.c</td>
<td>130</td>
</tr>
<tr>
<td>B.7.1</td>
<td>draw_mapped_curve</td>
<td>130</td>
</tr>
<tr>
<td>B.7.2</td>
<td>display_all_mapped_curves</td>
<td>131</td>
</tr>
<tr>
<td>B.8</td>
<td>draw_patch.c</td>
<td>132</td>
</tr>
<tr>
<td>B.8.1</td>
<td>draw_patch</td>
<td>132</td>
</tr>
<tr>
<td>B.8.2</td>
<td>draw_u_line</td>
<td>132</td>
</tr>
<tr>
<td>B.8.3</td>
<td>draw_w_line</td>
<td>133</td>
</tr>
<tr>
<td>B.9</td>
<td>draw_temp_struct.c</td>
<td>134</td>
</tr>
<tr>
<td>B.9.1</td>
<td>display_temp_struct</td>
<td>134</td>
</tr>
<tr>
<td>B.9.2</td>
<td>draw_transformed_curve</td>
<td>134</td>
</tr>
<tr>
<td>B.9.3</td>
<td>draw_projected_lines</td>
<td>135</td>
</tr>
<tr>
<td>B.10</td>
<td>map_curve.c</td>
<td>136</td>
</tr>
<tr>
<td>B.10.1</td>
<td>get_map_data</td>
<td>136</td>
</tr>
<tr>
<td>B.10.2</td>
<td>get_bicubic_data</td>
<td>136</td>
</tr>
<tr>
<td>B.10.3</td>
<td>map_intersect</td>
<td>137</td>
</tr>
<tr>
<td>B.10.4</td>
<td>get_edge_point</td>
<td>138</td>
</tr>
</tbody>
</table>
B.10.5  check_corner_intext .................................................. 138
B.10.6  join_edge ................................................................. 139
B.11  Model_curve.c .............................................................. 140
  B.11.1  Create_NUBS3_Curve .............................................. 140
B.12  process_picked_patch.c .................................................. 141
  B.12.1  process_picked_patch ............................................... 141
  B.12.2  create_patch_marker .............................................. 141
  B.12.3  display_picked_patch ............................................. 142
  B.12.4  new_marker_pt ..................................................... 143
  B.12.5  move_marker_pt ................................................... 144
B.13  ptr_val.c ................................................................ 145
  B.13.1  draw_ptr_val .......................................................... 145
  B.13.2  get_ptr_val ............................................................ 146
B.14  read_curve.c ................................................................. 147
  B.14.1  read_curve .............................................................. 147
  B.14.2  clean_curve ........................................................... 147
  B.14.3  check_text ............................................................ 148
B.15  retrieve_patch_data.c ..................................................... 149
  B.15.1  retrieve_patch_data .................................................. 149
  B.15.2  retrieve_patch_knots ............................................... 149
  B.15.3  corner_middle_data ................................................. 150
  B.15.4  cal_patch_point ..................................................... 151
  B.15.5  get_tangents ......................................................... 152
B.16  transform_curve.c .......................................................... 153
  B.16.1  find_normal .......................................................... 153
  B.16.2  orient_curve ........................................................ 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.16.3</td>
<td>transform_curve</td>
<td>154</td>
</tr>
<tr>
<td>B.16.4</td>
<td>flip_curve</td>
<td>155</td>
</tr>
<tr>
<td>B.16.5</td>
<td>rotate_curve</td>
<td>156</td>
</tr>
<tr>
<td>B.17</td>
<td>util.c</td>
<td>157</td>
</tr>
<tr>
<td>B.17.1</td>
<td>normalize_vector</td>
<td>157</td>
</tr>
<tr>
<td>B.17.2</td>
<td>matrix_mult</td>
<td>157</td>
</tr>
<tr>
<td>B.17.3</td>
<td>find_nearest_point</td>
<td>158</td>
</tr>
<tr>
<td>B.17.4</td>
<td>interpolation</td>
<td>159</td>
</tr>
<tr>
<td>B.17.5</td>
<td>unnormalize</td>
<td>159</td>
</tr>
<tr>
<td>B.17.6</td>
<td>find_nearest_edge_knot</td>
<td>160</td>
</tr>
<tr>
<td>B.17.7</td>
<td>find_near_param</td>
<td>161</td>
</tr>
<tr>
<td>B.17.8</td>
<td>minimum</td>
<td>161</td>
</tr>
<tr>
<td>B.17.9</td>
<td>maximum</td>
<td>162</td>
</tr>
<tr>
<td>B.17.10</td>
<td>traverse_point_data</td>
<td>163</td>
</tr>
</tbody>
</table>

Vita ........................................................................................................ 164
List of Illustrations

Figure 1  ACSYNT B-Spline Module Screen Layout................................................. 13
Figure 2  Cubic B-spline Control Polygon and Curve. ........................................... 18
Figure 3  Convex Hull Property of B-Spline Curves. ............................................. 20
Figure 4  Local Control Property of B-Spline Curves. .......................................... 21
Figure 5  Effect of Adding Multiple Knots to a Curve........................................... 26
Figure 6  Bicubic B-Spline Control Net and Surface........................................... 31
Figure 7  Intersection of a Line and a Surface................................................... 34
Figure 8  Untrimmed and Trimmed Surfaces...................................................... 40
Figure 9  Trim Curve Examples. ........................................................................ 42
Figure 10 Approximating a Trim Curve Using Surface Iso-parametric Line
          Segments ........................................................................................................ 45
Figure 11 Component and Model Data Structures ............................................... 49
Figure 12 Patch Data Structure and Added Entries in Component Data
          Structure. ........................................................................................................ 50
Figure 13 Surface Patches Numbering Scheme. .................................................... 52
Figure 14 ACSYNT Logo - A Mapping Curve Example......................................... 55
Figure 15 Selected Component Displayed as Individual Patches. ....................... 57
Figure 16  User Interface in Selecting Reference Mapped Point. ........................................ 58
Figure 17  The Normal and Tangent Vectors on a Bicubic Patch. .................................... 61
Figure 18  Transformation of Plane Curve to Mapped Reference Point. ............................ 63
Figure 19  General Spatial Transformation. ..................................................................... 65
Figure 20  Multiple Intersections Between a Projection Line and a Surface/Path .................. 70
Figure 21  Example Cases of Out-of-Bounds Mapped Curves. ........................................ 73
Figure 22  Surface Perimeter Patches. .............................................................................. 76
Figure 23  Determination of Edge Parameter for a Side Patch. ....................................... 78
Figure 24  Determination of Edge Parameter for a Corner Patch. ................................... 79
Figure 25  Reverse Projection of Surface Corners to the Plane Curve. ............................. 82
Figure 26  Internal-External Test. .................................................................................... 83
Figure 27  An ACSYNT B-Spline Logo. .......................................................................... 85
Figure 28  An ACSYNT logo mapped onto the Fuselage of an Aircraft Model................. 86
Figure 29  A Mapped Model in Four Views. ................................................................. 87
Figure 30  An Out-of-Bounds Curve Covering One Surface Corner. ............................... 88
Figure 31  An Out-of-Bounds Curve Covering Three Surface Corners. ......................... 89
Figure 32  Multiple Mapped Curves on a Surface Model. ............................................... 90
Figure 33  Curve Mapped Out-of-bounds on a Curved Boundary of a Surface.............. 94
Figure 34  Sample Curve Data File. ............................................................................... 107
Figure 35  Curves Created Using Data File Shown in Figure 34 ...................................... 109
Figure 36  Screen Layout Showing Picked Patch, Mapping Coordinates and Mapping Curve. ........................................................................................................ 113
Figure 37  Original and Flipped Curve Model. .............................................................. 115
Figure 38  Rotation of Curve Model. .............................................................................. 118
Figure 39  Map Preview Shown in Four Views. ............................................................... 119

List of Illustrations
List of Tables

Table 1  Description of Curve Data File. ................................................................. 108
Table 2  Pointer Valuator Buttons and Their Associated Directions. ............... 112
1.0 Introduction

Graphics provides one of the most natural means of communicating with a computer [Fole90]. The information conveyed through pictures and visualization of data, objects and images provided by computer graphics is virtually indispensable in many engineering design or implementation processes. This is best described by the ancient Chinese proverb "A picture is worth a thousand words". However, the tools needed for the creation and generation of meaningful and accurate geometry of scientific and engineering objects to allow processing by a computer invariably involves the derivation of a set of governing mathematical equations and representation that fully describes the objects. An object can be a simple, well-defined geometry, such as a rectangular block or a cylinder, or a sophisticated geometry that can only be defined using a free-form surface. Computer Aided Geometric Design (CAGD) is the field concerned with the approximation and representation of curves and surfaces that arise when these objects have to be processed by a computer [Böhm84].

Geometric Modeling, which can broadly be defined as a collection of methods that are used to describe the shape and other geometric characteristics of an object [Mort85], is one of the key elements in Computer-Aided Design (CAD). For many CAD applications,
the design of surfaces is not sufficient by itself [Barn87]. The surfaces usually require further processing to achieve the desired product specifications. Examples of the additional features may be a fillet between two surfaces to smooth out the connection, intersections between two surfaces, or a curve on a surface which can be used as an added detail to the design or to treat the curve as a trimming curve when only a part of the designed surface is needed.

Recognizing the importance of graphical visualization in CAD and the geometric modeling of surface design, this research concentrates on designing curves on surfaces by the method of normal projection. The surfaces for mapping are parametrically defined bicubic B-spline representations, which are very popular in modeling engineering objects in CAD applications. The user interface and interactivity of the mapping process is incorporated into the B-Spline Module of ACSYNT - an interactive computer-aided conceptual aircraft design software jointly developed by NASA Ames Research Center and Virginia Tech CAD Laboratory. The curve mapping routine implemented in this PHIGS (Programmers Hierarchical Interactive Graphics System)-based menu-driven software provides full user interaction from curve generation by data points to mapping position and orientation of multiple curves.
2.0 Problem Definition, Research Objectives and Thesis Organization

2.1 Problem Definition

With the advent of high-speed workstations, computer processing power has increased dramatically in the past few years. This has greatly benefited the power and speed-hungry fields of computer-aided design and computer graphics, which require high-speed calculations and the generation of complex objects represented by a set of complicated mathematical formulae. With the availability of the powerful workstations, free-form models can now be processed beyond the level of mere geometrical description and display. The design of curves on surfaces is one example. The problems posed by the design of curves on well-defined surfaces and its directly associated problems are manifold. In a simple case, the curve can consist of a set of discrete points that are guaranteed to be lying on the surface (such as explicitly defining the points in terms of surface parameters). The only requirement is to find a function to interpolate the points. In another case, the curve can be a plane or space curve defined in its own modeling coordinates to be mapped onto the design surface. This mapping process inherently
produces its own problems. Projecting the curve normally (perpendicularly) onto the surface is analogous to producing the shadow of an object on a curved surface by shining a light at the object is one method. Another plausible method is by 'wrapping' the curve onto the contour of the surface, with the analogy of a soap bubble collapsing on a surface. The projection approach in turn poses the need for finding intersections. With a third-degree (cubic) parametric surface, finding intersections requires high-speed processors for computing numerical solutions. In any case, guaranteeing that all mapped points be exactly on the design surface is critical for a meaningful design. The next step is the representation of the curve using the mapped points. Fitting a spline through the points produces a smooth curve. This is useful for design visualization, such as visually determining the placement of control surfaces (flaps, ailerons and spoilers) on aircraft wings, interference checks, etc., or for geometrical (mathematical) trimming. The major problems associated with geometrically trimming a surface are in representing the resulting trimmed surface as tensor product polynomial surface patches (see Chapter 6). An innovative way of representing the trim curve that will result in a trimmed surface which can still be described fully by tensor product polynomials is highly desirable and will tremendously simplify the surface trimming. Lastly, the problem associated with the design of curves on surfaces is encountered when the curve is going out-of-bounds on the surface. Surface boundary and corner information is needed in this case to close the curve. This seemingly simple problem can turn out to be monstrous in the number of different cases possible (see Chapter 8).
2.2 Research Objectives

With the number of problems presented, solving each one in this thesis is virtually impossible. Several sampled topics will be addressed. The objectives of this research are outlined below:

- Develop a data file for the creation and display of B-spline closed curves.

- Create a user interface for the definition of mapped points on the surface for normal curve projection.

- Investigation of ways of representing mapped curves for the purpose of visualization and storing point information in a data structure that could be useful for geometrical trimming of the surface.

- Investigation of ways of determining the criteria for getting appropriate surface boundary and/or corner information of the mapping surface should the mapped curve be out-of-bounds.

Fleming's constraint-based inversion routine [Flem92] and Feustel's line-bicubic surface intersection routine [Wong90] [Wong91] [Mykl91] are employed in this research.
2.3 Thesis Organization

The organization of this thesis directly follows the course taken in conducting the research. Chapter 3 surveys the literature reviewed on topics related to this research. Chapter 4 introduces the ACSYNT B-Spline Module. Chapter 5 presents a brief description of parametric B-spline curves and surfaces, Fleming's inversion routine for curve creation, as well as the intersection routine. Chapter 6 discusses the tensor product surface and its relation to the problem of surface trimming. Chapter 7 presents the details on the mechanism, steps, user interface and interaction in the curve mapping process. Chapter 8 discusses the steps taken in getting correct surface boundaries and/or corner information for out-of-bounds mapped curves. Results, conclusion and references follow next. A user guide for curve creation, data files and the use of the Curve Mapping Module in the ACSYNT B-Spline Module is included in Appendix A, while C source code functional description is included in Appendix B.

In the context of this research, due to the close relationship of the processes, the words mapping and projection are often used interchangeably.
3.0 Literature Review

Computer-aided design (CAD) and computer graphics, though relatively new, have been very important in developing engineering applications. With the advent of high-speed computer workstations, more advanced computer-aided modeling tools can be developed to aid the increasingly demanding hi-tech design processes. However, modeling real-world objects is an extremely complex process. This is especially true in the design of free-form curves and surfaces. Research literature in this area is voluminous, but not nearly enough to cover all areas. Zieg presents a comprehensive survey on the theory and practice of the CAD/CAM concepts [Zieg91], while Foley et al. discuss the complex field of computer graphics and its principles and practice [Fole90].

Geometric modeling is one of the most important CAD tools. Mortenson devotes a whole book to discussing the concepts and underlying mathematical techniques of geometric modeling [Mort85].

Farin states that parametric curves and surfaces can be regarded as the origin of Computer Aided Geometric Design (CAGD) [Fari90]. While Farin and Yamaguchi present the basics to advanced concepts of curves and surfaces for use in CAGD [Fari90] [Yama88],
Böhm et al. give a definition of CAGD and a survey of various curve and surface methods [Böhm84]. Other excellent materials that discuss the various aspects and methods of curve and surface design can be found in Faux and Pratt, and Bartels et al. [Faux79] [Bart87]. Piegl discusses the key developments in CAGD [Piegl89].

B-spline geometry has been discussed extensively in the above literature. It is regarded to be the most suitable mathematical model to represent free-form features. The underlying controlling elements of B-spline geometry, the basis functions, were derived and presented by de Boor, Yamaguchi and Bartels et al. [deBo72] [Yama88] [Bart87].

Barnhill et al. state that the design of surfaces is usually not sufficient by itself [Barn87]. Certain manipulation schemes, for example, intersections and trimming, must be imposed on the design surfaces to achieve the final product specifications. Numerical techniques such as the Newton-Raphson iteration scheme are often employed in the intersection problem. Certain subdivision techniques such as divide-and-conquer and marching techniques are also used [Peng84] [Barn87] [Barn90a]. Wong et al. developed an analytical and numerical approach for finding intersection curves of two bicubic B-spline surfaces by elimination methods [Wong90] [Wong91]. They present the idea of approximating the B-spline surfaces with ruled surfaces to be analytically manageable. This approach also increases the accuracy significantly in comparison to the traditional divide-and-conquer methods, which approximate the surface with planar polygons. Feustel later implemented Wong's idea in improving codes for surface-surface intersections [Mykl91]. Chiang and Wilde present a method to find parametric bicubic surface intersections which ultimately reduces the problem to finding the intersection of a series of spatial cubic parametric curves [Chia89]. The cornerstones of their algebraic method are the formulation of a bicubic curve by Sylvester expansion and Bezout's
Resultants Method for finding the common roots of two polynomials, however, solution examples are not presented in the paper.

The other surface processing procedure is trimming. Howard and Gaskins present the visual trimming implementation in PHIGS Plus [Howa91] [Gask92]. Hoschek and Schneider developed an approximation procedure for the conversion of rational B-spline surfaces of arbitrary order into a bicubic set of integral Bézier patches [Hosc90]. Since polynomial surfaces are special cases of the rational surfaces, the method presented in the paper is also valid for polynomial trimmed surfaces. Sheng and Hirsch present a method to triangularize trimmed surfaces in parametric space [Shen92].

Designing curves on surfaces is a very important aspect of CAD and computer graphics. Pegna and Wolter discuss its importance in the design and manufacturing of shell structures for construction of trimming curves on surfaces [Pegn90]. They devised an approach of designing and mapping trimming curves on surfaces using orthogonal projection, which is an extension to the distance projection (the finding of the closest points in a surface to a given point in space). Each point on the space curve is mapped onto the surface by finding its closest distance to the surface via the surface normal. Consequently, the space curve has to remain close enough to the surface and the projected curve has to be in the interior of the surface patch. The mapped surface curve is then solved numerically by solving an initial value problem by the marching method. This is in contrast to the work done in this thesis research which addresses a normal projection of a plane curve onto a B-spline surface in which all points on the curve are projected and mapped onto the surface in one direction, which is chosen to be the normal to the plane. The surface points are found by intersections using an elimination method. The mapping curve generally does not have to be completely interior to an open surface. Barnhill and Ou present two methods for constructing functions defined over convex
surfaces for the purpose of visualization [Barn90b]. These modified Shepherd's and triangular-based methods interpolate data values at scattered points on the surface. Data points are assumed to be given and the mapping surfaces have to be convex.

The display is the most important aspect in computer graphics. Standards have been implemented by organizations such as the American National Standards Institute (ANSI) and the International Standards Organizations (ISO). PHIGS - the Programmer's Hierarchical Interactive Graphics System, is the ISO standard for 3-D graphics applications programming interfaces (APIs). The most important feature of the PHIGS standard is its device independence. Jayaram et al. propose an approach towards the establishment of a CAD/CAM programming standard based on PHIGS [Jaya89] [Jaya90] [Jaya93a]. Lin et al. developed a working prototype for a new and unique object-oriented software development environment to integrate the design geometry with the constraints imposed by design processes [Lin92] [Lin93]. With the increased popularity of object-oriented programming, the idea of incorporating object-oriented class libraries for the creation of device independent, PHIGS-based modeling software has been developed [Mykl92] [Uhor93] [Woya93].

Several PHIGS-based systems and software have been developed. Thatch and Myklebust describe a system for interactively designing spatial mechanisms using PHIGS [That88]. Wampler et al. designed a completely PHIGS-based computer-aided conceptual aircraft design software - ACSYNT [Wamp88], while Jones and Marcaly extended the B-spline geometry module to ACSYNT [Jone91] [Marc91]. Several enhancement features have been added onto ACSYNT, such as intersection, filletting [Glou89] [Glou90] [Jone91] and dimensional geometric parameter extraction from B-spline surface models [Jaya91] [Jaya92b].
4.0 ACSYNT

4.1 ACSYNT

ACSYNT (AirCraft SYNThesis) is a conceptual aircraft design code developed by NASA Ames Research Center in the early 1970's. A CAD system for ACSYNT has been under development since 1987 at the Virginia Tech CAD laboratory [Wamp88]. It was designed and coded entirely with the three-dimensional graphics standard, PHIGS, for the Graphical User Interface (GUI) and the geometry rendering to achieve machine and graphics device independence.

ACSYNT CAD geometry consists of parametrically generated surface models. Each component is described by a set of equations, a set of characteristic points, and a corresponding set of parametric cross-sections that are lofted through with bicubic B-spline patches [Jaya92a]. The basis for the modeling structure is partially discussed in [Wamp88] [Jone91] [Jaya92a] [Jaya92c].
4.2 ACSYNT B-Spline Module

The ACSYNT B-Spline Module was created from software designed to serve as a development platform for testing new algorithms in surface and solid modeling [Jone91]. However, due to the increased popularity of B-spline surface models, and because of the popular demand from members of ACSYNT Institute, the B-Spline Module has been integrated into ACSYNT main code from version 2.0 on [Acsy93].

ACSYNT B-spline models are created using non-uniform B-spline surfaces. Cross-sectional data points can be inverted to get the corresponding control points and knot sequences [Glou89] [Glou90]. Currently, the module supports capabilities such as displaying curvature plots, determining intersection curves and filleting between different components, and modification of the surfaces using handles. Another important feature is its parameter extraction capability, in which dimensional geometric parameters of the aircraft such as wing span, fuselage fineness ratio, moments of inertia, etc. can be extracted from a modified model [Jaya91] [Jaya92b]. This is very important in an iterative design process.

This curve projection and representation research has been tested and implemented in the ACSYNT B-Spline Module version 1.2.2. An ACSYNT B-Spline Module version 1.2.2 screen layout and its corresponding windows are shown in Figure 1.
Figure 1  ACSYNT B-Spline Module Screen Layout.
5.0 Parametric B-Splines

5.1 Cubic Parametric Equations

A mathematical representation is required to model real-world objects, which are then represented by the computer in CAD applications to generate the object. Much of the power of computer graphics and geometric modeling lie in their ability to accurately model these real world objects, which are inherently smooth. Apart from being smooth, curves describing engineering objects are generally well-behaved. The treatment of these curves in computer graphics and CAD applications is different from that in analytical geometry or approximation theory. Furthermore, not every available form of a curve equation is efficient to use in CAD software due to either computation or programming problems [Zeid91].

Parametric equations of the form:

\[
\begin{align*}
  x &= x(t) \\
  y &= y(t) \\
  z &= z(t)
\end{align*}
\]
offer attractive features in computer programming. The most common and useful parametric polynomial is of the third degree (cubic). The cubic polynomial gains its popularity for several reasons, two of which are described below:

1. Lower degree (lower than 3rd) polynomials provide insufficient control of the curve shape, while higher degree polynomials introduce too many undesirable fluctuations and are computationally more difficult.

2. Parametric cubics are the lowest degree curves that can be non-planar.

The algebraic form of a parametric cubic (PC) curve segment in vector notation is given by:

\[ \mathbf{P}(u) = \sum_{i=0}^{3} C_i u^i \]

\[ 0 \leq u \leq 1 \]  

(5.2)

or in the expanded Cartesian form:

\[ x(u) = C_{3x} u^3 + C_{2x} u^2 + C_{1x} u + C_{0x} \]

\[ y(u) = C_{3y} u^3 + C_{2y} u^2 + C_{1y} u + C_{0y} \]

\[ z(u) = C_{3z} u^3 + C_{2z} u^2 + C_{1z} u + C_{0z} \]  

(5.3)

The set of 12 constant coefficients, called the algebraic coefficients, determines a unique parametric cubic curve - its size, shape and position in space.

Cubic spline equations will be developed and discussed later in this chapter.

For the above reasons, all curves and surfaces implemented in this research are parametric cubic polynomials of a specific form called B-spline curves and surfaces.
5.2 B-Splines

The letter 'B' in B-splines stands for Basis. B-splines are piecewise polynomial splines governed by a set of basis (blending) functions. A recursion relationship for the basis function has been independently found by Mansfield, de Boor and Cox [Faris89]. The expression is shown below:

\[ N_i^k(u) = (u-t_i) \frac{N_i^{k-1}(u)}{t_{i+k} - t_i} + (t_{i+k+1} - u) \frac{N_{i+1}^{k-1}(u)}{t_{i+k+1} - t_{i+1}} \]  
\[ \text{where} \quad N_i^1 = \begin{cases} 1, & u \in [t_i, t_{i+1}] \\ 0, & u \notin [t_i, t_{i+1}] \end{cases} \]

\( N_i^1(u) \) : the \( i \)th basis function of order \( k \) (degree \( k-1 \)) as a function of parameter \( u \).

\( t_i \) : knots or knot values.

Besides being recursive, the basis function exhibits the following properties:

1. **Partition of Unity:** \[ \sum_{i=0}^{k} N_i^k(u) = 1 \]
2. **Positivity:** \[ N_i^k(u) \geq 0 \]
3. **Local Support:** \[ N_i^k(u) = 0 \quad \text{if} \quad u \notin [t_i, t_{i+k+1}] \]
4. **Continuity:** \( N_i^k(u) \) is \((k-2)\) times continuously differentiable.

Since \( N_i^1 \) is constant for \( k=1 \), a general value of \( k \) produces a polynomial in \( u \) of degree \((k-1)\) (Eq. 5.4) and therefore a curve of order \( k \) and degree \((k-1)\). The knot
values form a set of non-decreasing real numbers called the knot sequence. \( k + 1 \) knots (\( k \) intervals) are needed to define a basis function, and this function is dependent on the defining knot spacing, not the actual knot values.

Fleming has a more detailed description of the B-spline basis functions [Flem92], while Yamaguchi [Yama88, pp. 285-291] and Bartels et al. [Bart87] provide rigorous derivations.

### 5.3 B-Spline Curves

The weighted combination of the governing B-spline basis or blending functions forms the corresponding B-spline curves, which are piecewise polynomials. A B-spline curve is defined in terms of the locations of a set of points equal in number to the order (degree + 1) of the curve. These points are called the control or de Boor points. They form the vertices of the control or characteristic polygon, which uniquely defines the curve shape, as shown in Figure 2 for the cubic case.

Mathematically, for \( n + 1 \) control points, the B-spline curve is defined by the following parametric polynomial of degree \( n \):
Figure 2  Cubic B-spline Control Polygon and Curve.
\[ P(u) = \sum_{i=0}^{n} P_i N_i^k \]
\[ n \geq k - 1 \]

For a cubic B-spline, \( n = 3 \).

\( N_i^k(u) \) are the B-spline basis functions as shown in equation 5.4. The \( P_i \)'s are the control points. The order \( k \) controls the degree \((k - 1)\) of the resulting B-spline curve and is not affected by the number of control points.

The first three properties of the B-spline basis function shown on page 16 affect the relationship between the curve and its defining control points [Zeid91]. The *Partition of Unity* property ensures that the relation between the curve and its control points is invariant under affine transformation. The *Positivity* property guarantees the *convex hull* property. A curve is said to have the convex hull property if it lies entirely within the convex hull defined by the polygon vertices. Figure 3 shows the convex hull property. The hull is formed by wrapping a piece of rubber-band around its vertices. Finally, the *Local Support* property is perhaps the most important of all. It indicates that each segment of a B-spline curve is influenced by only \( k \) curve segments. Thus, for a cubic B-spline \((k = 4)\), each segment is affected by at most four control points, and each control point influences only four curve segments (Figure 4). Consequently, the time needed to compute the coefficients is greatly reduced.
Figure 3  Convex Hull Property of B-Spline Curves.

Parametric B-Splines
Figure 4  Local Control Property of B-Spline Curves.
The local control property brings out the interesting nature of the B-spline formulation, which has commonly been described as both a single cubic polynomial segment as well as a set or sequence of cubic polynomials. The general advantage of a sequence of B-spline cubic polynomials is, however, in its ability to increase the number of control points without raising the degree of the curve. Thus moving a control point of a piecewise or composite B-spline curve affects only a few segments of the curve. Consequently, Bartels, et al [Bart87] point out that local control should only be referred to when considering a composite B-spline curve.

A B-spline curve can be either uniform or non-uniform, depending on the spacing of the knot sequence. If the spacing between all knots is the same, the curve is uniform. For a uniform B-spline, the blending functions for each curve segment are identical, because for each segment \( i \) the values of \( u - u_i \) range from 0 at \( u = u_i \) to 1 at \( u = u_{i+1} \).

If the spacing between successive knots is different, the curve is non-uniform. This implies that the blending functions for each interval are no longer the same. The advantages of these curves over their uniform counterparts are:

- Continuity between curve segments can be reduced, for a cubic curve, from \( C^2 \) to \( C^1 \) to \( C^0 \) to none without the undesirable effect of curve segments on either side of the interpolated control point (when continuity is reduced to \( C^0 \)) becoming straight lines [Fole90].

- Starting and ending points can also be interpolated without the straight line effects.

- Additional knots and control points can be added to provide further shaping capabilities.
Parametrization is the technique of determining an appropriate knot spacing. Besides Uniform, other parametrization techniques include Centripetal, Chord Length, and Foley. Farin investigates these techniques in detail [Fari90].

5.4 Constraint-Based B-Spline Curve Inversion

5.4.1 Curve Inversion

A curve inversion or inverse transformation process involves finding the set of the defining control points given the set of data points to be interpolated. The B-spline inversion process typically consists of setting up and solving a system of linear equations based on interpolation conditions required by model data [Flem92]. Yamaguchi describes the inversion process for a uniform cubic B-spline [Yama88] while Gloudemans extended the process to a non-uniform case [Glou89].

The inversion process typically involves solving a set of simultaneous equations. Normally, curve end conditions are needed to supply sufficient constraints to solve the system of equations. These end conditions are typically expressed in terms of a zero curvature or a tangency condition for an open curve. A closed curve does not require these end conditions. However, ending control points at both ends of the curve have to overlap in such a fashion that the last point coincides with the second and the second to
the last coincides with the first to ensure $C^2$ continuity. It is worth mentioning that by simply coinciding the first and last control points of an open curve generates only a $C^0$ continuous closed curve, while making the first and last segment of the control polygon collinear produces a $C^1$ continuous closed curve.

In an inversion process, solving for the control points typically involves solving the system of equations shown below [Fari90]:

\[
\begin{align*}
\begin{bmatrix}
  \text{end} \\
  d_1 \\
  d_2 \\
  \vdots \\
  d_n \\
  \text{end}
\end{bmatrix}
\begin{bmatrix}
  N^4_0(u_s) & N^4_1(u_s) & N^4_2(u_s) \\
  N^4_1(u_s) & N^4_2(u_s) & N^4_3(u_s) \\
  \vdots & \vdots & \vdots \\
  \vdots & \vdots & \vdots \\
  N^4_{n-1}(u_{s+2}) & N^4_n(u_{s+2}) & N^4_{n+1}(u_{s+2})
\end{bmatrix}
\begin{bmatrix}
  0 \\
  p_0 \\
  p_1 \\
  p_2 \\
  \vdots \\
  p_{n+1}
\end{bmatrix} = \mathbf{D} = \mathbf{NP}
\end{align*}
\]

\(D = \text{data points}\)

\(N = \text{basis function values at the knots}\)

\(P = \text{control points}\)
5.4.2 Constraint-Based Inversion Technique

As stated in section 5.2, the B-spline basis function is \((k - 2)\) times differentiable as long as there are no multiple knots. This means that a cubic curve is guaranteed to have \(C^2\) as its highest order continuity everywhere along the composite curve. It also has the ability to represent \(C^1\), \(C^0\), or even no continuity at all in its geometries.

Fleming [Flem92] presents a novel approach to incorporating continuity constraints into cubic B-spline geometry. The idea utilizes the sensitivity of the B-spline basis functions to multiple knots. The general behavior of a basis of order \(k\) is such that at a knot of multiplicity \(m\), the B-spline exhibits \(C^{k-m-1}\) continuity. This behavior shows that the multiple knots will reduce parametric continuity: from \(C^2\) to \(C^1\) for one extra knot or a multiplicity of 2; from \(C^1\) to \(C^0\) for two extra knots or a multiplicity of 3; and to carry one step further which is not incorporated in Fleming's technique; from \(C^0\) to none for three extra knots or a multiplicity of 4. Figure 5 shows the effect of adding multiple knots to a curve.

The concept of multiple knots and their effects on continuities have been incorporated into the B-spline inversion process by Fleming. The process involves the following steps:

1. Parametrize data points based on the appropriate technique (uniform, chord length, etc.)

2. Add additional knots into the knot sequence at knot values where the continuity should be reduced. Add one additional knot for \(C^1\) or two for \(C^0\).
(a) No multiple knots - knot sequence (0, 1, 2, 3, 4, 5)

(b) Double knots - knot sequence (0, 1, 1, 2, 3, 4)

(c) Triple knots - knot sequence (0, 1, 1, 2, 3)

(d) Quadruple knots - knot sequence (0, 1, 1, 1, 2)

Legend:  
- **P** - Control points  
- **Q** - Spline segments  
- **t** - Knots  
- **-** - Control Hulls

Figure 5  Effect of Adding Multiple Knots to a Curve.
3. Evaluate appropriate basis function values and derivatives at all domain knot values.

4. Assemble the linear system including tangent vector constraints and basis function derivatives for all $C^1$ and $C^0$ points.

5. Solve the system of equations.

The tangent vectors in step 4 act as the extra constraints needed to solve for the new control point introduced each time a knot is added in the inversion process. For $C^1$ continuity, one tangent vector is required, and for $C^0$ continuity, two tangent vectors are required.

The beauty of the above-described process is that a designer is able to create a B-spline curve intuitively by providing the inversion routine with the necessary information such as the coordinates of a minimum number of data points that are to be interpolated by the curve, with the continuities desired at the interpolated points and the constraints. The curves created in this research for mapping purposes are created by such a process.
5.5 **Parametric Bicubic Surfaces**

A surface is a collection of points whose coordinates are given by a continuous, two-parameter, single valued mathematical function of the form:

\[
x = x(u,w) \\
y = y(u,w) \\
z = z(u,w)
\]  

(5.8)

which is a second parametric dimensional extension of equation 5.1. If the parameters are constrained to a finite interval, a bounded surface is produced.

The simplest form of the mathematical element to model a surface is a *patch*. The algebraic form of a bicubic (cubic in both parametric directions) patch is given by:

\[
S(u,w) = \sum_{i=0}^{3} \sum_{j=0}^{3} C_{ij} u^i w^j \\
0 \leq u \leq 1, \quad 0 \leq w \leq 1
\]

(5.9)

or in the expanded Cartesian form:

\[
x(u,w) = C_{33} u^3 w^3 + C_{32} u^3 w^2 + C_{31} u^3 w + C_{30} u^3 + \\
C_{23} u^2 w^3 + C_{22} u^2 w^2 + C_{21} u^2 w + C_{20} u^2 + \\
C_{13} u w^3 + C_{12} u w^2 + C_{11} u w + C_{10} u + \\
C_{03} w^3 + C_{02} w^2 + C_{01} w + C_{00}
\]

(5.10)
\[ y(u,w) = C_{33}, u^3 w^3 + C_{32}, u^3 w^2 + C_{31}, u^3 w + C_{30}, u^3 + \\
C_{23}, u^2 w^3 + C_{22}, u^2 w^2 + C_{21}, u^2 w + C_{20}, u^2 + \\
C_{13}, uw^3 + C_{12}, uw^2 + C_{11}, uw + C_{10}, u + \\
C_{03}, w^3 + C_{02}, w^2 + C_{01}, w + C_{00}, \\
\]

\[ z(u,w) = C_{33}, u^3 w^3 + C_{32}, u^3 w^2 + C_{31}, u^3 w + C_{30}, u^3 + \\
C_{23}, u^2 w^3 + C_{22}, u^2 w^2 + C_{21}, u^2 w + C_{20}, u^2 + \\
C_{13}, uw^3 + C_{12}, uw^2 + C_{11}, uw + C_{10}, u + \\
C_{03}, w^3 + C_{02}, w^2 + C_{01}, w + C_{00}, \\
\]

Similar to the parametric cubic (pc) curve, this set of 48 algebraic coefficients uniquely determines the parametric bicubic surface in space.

This algebraic form of the surface will be used for line-surface intersection discussed later in the thesis.
5.6 **B-Spline Surfaces**

A B-spline surface is the generalized two-dimensional formulation of the B-spline curve. It is formed by a weighted combination of the governing two-dimensional B-spline basis or blending functions. A rectangular set of control points forms the vertices of the control polyhedron that approximates and controls the shape of the resulting surface. The control net and the resulting bicubic B-spline surface is shown in Figure 6.

A B-spline surface patch defined by an \((n+1) \times (m+1)\) array of control points is given by extending equation 5.6 into two dimensions:

\[
S(u,w) = \sum \sum P_{ij} N_i^k(u) N_j^l(w) \tag{5.11}
\]

where

- \(P_{ij}\) = control points
- \(N_i^k(u) = \) \(i\)th basis function of order \(k\) as a function of \(u\)
- \(N_j^l(w) = \) \(j\)th basis function of order \(l\) as a function of \(w\)

All the related discussions to the B-spline basis in section 5.2 apply to the above equation.

B-spline surfaces have the same characteristics as B-spline curves. \(C^2\) continuity across boundaries is automatic; no special arrangements of control points are needed except to avoid duplicates, which create discontinuities.
Figure 6  Bicubic B-Spline Control Net and Surface (Adapted from [Flem92]).
An interesting characteristic to observe in this form of rectangular net surface, which is known as a tensor product surface (see Chapter 6), is the iso-parametric lines, in which each is a B-spline curve on its own. For instance, lines of constant \( w \) are B-spline curves with de Boor points

\[
P_i(u) = \sum_{i=0}^{s} p_i N_i^s(u)
\]

(5.12)

Analogous to the B-spline curve, a B-spline surface is compositely made up of polynomial patches. The local control of a bicubic B-spline surface is affected by 16 control points, as shown by the following matrix equation for a non-uniform bicubic B-spline:

\[
S_{ij} = \begin{bmatrix}
N_0^s(u) & N_1^s(u) & N_2^s(u) & N_3^s(u)
\end{bmatrix}
\begin{bmatrix}
p_{i-1,j-1} & p_{i-1,j} & p_{i-1,j+1} & p_{i-1,j+2} \\
p_{i,j-1} & p_{i,j} & p_{i,j+1} & p_{i,j+2} \\
p_{i+1,j-1} & p_{i+1,j} & p_{i+1,j+1} & p_{i+1,j+2} \\
p_{i+2,j-1} & p_{i+2,j} & p_{i+2,j+1} & p_{i+2,j+2}
\end{bmatrix}
\begin{bmatrix}
N_0^s(w) \\
N_1^s(w) \\
N_2^s(w) \\
N_3^s(w)
\end{bmatrix}
\]

(5.13)

Gloudemans' non-uniform inversion technique [Glou89] includes B-spline curves and surfaces and is incorporated in the ACSYNT B-Spline Module to convert characteristic point data to B-spline surfaces. Fleming also extended his constraint-based inversion routine to surfaces [Flem92].
5.7 Projection

Projecting a curve onto a surface is useful in applications such as the detail and enhancement of surface designs, determining shadows or finding the position of the curve relative to the surface. The projection problem addressed here boils down to the problem of projecting successive data points on the plane curve to the surface. Taking only one point, the problem is shown in Figure 7. Point $P_o$ is projected along the direction $r$ onto the mathematically defined bicubic B-spline surface. The equation of the projected line is given by:

$$ P(t) = P_o + t\hat{r} \quad (5.14) $$

or, if two points are given instead of the directional vector $\hat{r}$, the equation of the line can be taken to be of the following Cartesian or scalar form:

$$ x(t) = (1.0 - t)p_{0x} + tp_{1x} \quad (5.15) $$
$$ y(t) = (1.0 - t)p_{0y} + tp_{1y} $$
$$ z(t) = (1.0 - t)p_{0z} + tp_{1z} $$

The equation of the surface is as shown in equation 5.9 or 5.10 with all forty eight coefficients known. The projection point $Q$ is the intersection point between the line and the surface. Thus, equations 5.10 and 5.15 are equated to produce 3 equations in 3 unknowns ($t, u$ and $w$).
Figure 7  Intersection of a Line and a Surface.
Solving the intersection problem involving surfaces is complex and nonlinear in nature. Numerical methods such as the Newton-Raphson iterative scheme or some subdivision algorithms such as divide-and-conquer techniques are usually employed in such a process. However, numerical techniques often fail mysteriously in solving the intersection equations. Subdivision algorithms, in which the surfaces are usually approximated with planar polygons, is computationally expensive and often results in the reduction in accuracy due to the planar polygonal approximations [Wong90]. In his thesis, Wong created a new algorithm which solves for the intersection curves between two bicubic B-spline surfaces using analytical elimination methods, where the bicubic B-spline surfaces are approximated using ruled surfaces [Wong90]. Later, Feustel implemented a subset of the surface-surface intersection algorithm to line-bicubic surface intersection [Mykl91]. The algorithm developed solves, in essence, the simultaneous equations presented above, which are equations 5.10 and 5.15. It is this developed code that was employed in finding the intersections of the mapping process. Some post-intersection processes such as determining the correct intersection if multiple points are found are needed in the curve mapping process. These will be discussed in Chapter 7.
6.0 Tensor Product Surfaces, Trimming, and Curve Representation

6.1 Background

It has been suggested in the last chapter that a general three-dimensional surface can be modeled by assembling a network of patches, which is considered as the basic mathematical element to model a composite surface. The topology of a patch may be, for example, \textit{n-sided polygons}, \textit{triangular} or \textit{rectangular}. Triangular patches are most widely used in finite-element analysis as mesh elements. Because of this, the triangular approach has gained a considerable amount of attention and recognition in CAD research. However, today's CAD systems utilize surfaces that are almost completely of the rectangular type [Fari89]. This trend can be traced back to the origin of surface design applications, which are the design of outer car panels and airplane fuselages, which possess intrinsically rectangular structures.

The problems associated with geometrical or mathematical surface trimming are mainly in the difficulties in explicitly expressing the trimmed surface mathematically. This is because irregular boundaries will most probably be generated after trimming procedures.
Specifically, a rectangularly defined design surface might not be rectangular in topology anymore after a portion has been trimmed off. This means mathematically defining the new surface will impose problems.

6.2 Tensor Product Surfaces

In surface modeling, a rectangular mathematical patch is called a tensor product, or Cartesian product patch. Its widespread use can generally be attributed to several reasons:

1. Its simple separable nature involving only products of univariate basis functions, usually polynomials.

2. Its formulation is simply a two-dimensional extension or generalization of that of a curve, which means it introduces no new conceptual complications.

3. Its properties can easily be deduced from properties of the underlying curve schemes.

4. It fits naturally onto rectangular topologies.

5. It has an explicitly unique orientation and special parametric or coordinate directions associated with each independent parametric variable.
Böhm et al. give a derivation of a tensor product surface [Böhm84], which results in an equation of the form:

$$S(u, w) = \sum_i \sum_j P_{ij} F_i(u) G_j(w)$$  \hspace{1cm} (6.1)

where $S(u, w)$ is the surface and the products $F_i(u) G_j(w)$ are blending functions for the surface.

Comparing equation 6.1 with that of the B-spline surface in equation 5.11, it is clear that a B-spline surface is a kind of tensor product surface. The matrix form of a B-spline formulation in equation 5.13 can be more concisely written as:

$$S(u, w) = N(u) P N^T(w)$$  \hspace{1cm} (6.2)
6.3 Surface Trimming

As stated earlier, design surface is seldom complete in its own right. Manipulating the surface in order to achieve the final product specification is often necessary. One such surface manipulation is trimming. Trimming of surface entities is useful for many engineering applications, especially in surface analysis. Figure 8 shows a surface before and after trimming.

Surface trimming can be of two categories: visual, or sometimes known as graphical trimming, or as "trimmed surfaces", and geometrical, or mathematical trimming. As suggested by the name, visual trimming is a technique which trims the unwanted portion off by not showing it. The mathematical representation of the surface is unchanged, which means that the equations for both the original and trimmed surfaces are the same. The trimmed portion is simply not rendered and thus is similar to a 'hidden' effect.

Visual trimming has its advantages as well as disadvantages. It is useful when analysis is not needed to be performed on the trimmed surface, such as interference simulation of dynamic processes, and the main purpose is to provide visual feedback on the look of the final product to the designer. Coupled with the comparative high speed in performing the process, visual trimming is important. This is true since in the conceptual design phase, many designs are usually created and drastic changes are made. Therefore, speed and intuitive look are the key factors.
Figure 8  Untrimmed and Trimmed Surfaces.
The main disadvantage of visually trimming a surface is in the incompleteness of the surface definition, or rather, 'over-definition' of the surface. Mathematically, the trimmed parts still exist, and thus analytic results cannot be achieved accurately on the surface. This is especially true in solid modeling, when intersections of two primitives mean that parts of one or both primitives have to be absolutely trimmed off.

Geometrical trimming overcomes the disadvantages of visual trimming. However, it poses some problems of its own. An original tensor product surface is mathematically tractable and easily manageable, whereas the trimmed surface with irregular trim-boundaries resulted after trimming operation is not. As shown by some examples in Figure 9, the resulting surface often cannot be represented by tensor product patches anymore, especially at the trim boundaries. A new scheme is needed to mathematically represent the surface. n-sided or triangular patches may be needed to model the patches at trim boundaries in addition to rectangular patches at untouched portions. This combination of different topological patches in one surface may be a major mathematical and programming problem if it is possible at all. If however, a way could be devised such that the remaining surface could still be represented by tensor product patches, the trimming problem might be greatly diminished.

Part of this research is trying to provide a unique way of such a representation of the surface. As stated in the introductory chapter, the curve being mapped onto the surface can be used for visual purposes to enhance the detail of a design surface or to serve as a new representation of trim curve boundaries. The first is a representation of a trimming curve for visual trimming and the second is a representation for geometrical trimming. Consequently, two separate sets of data structures have been developed and maintained.
6.4 Curve Representation

The parametric space of a tensor product surface provides a two-dimensional space suitable for representing an analytical curve. These parametric curves are normally used to represent trim curve boundaries on the surface. They are represented by a new parameter \( t \), such that

\[
\begin{align*}
u &= u(t) \\
w &= w(t)
\end{align*}
\]  

(6.3)

The data points of the curves can be obtained through intersections or be defined explicitly as surface parametric data, such as the trim curve function implemented in PHIGS Plus. This approach, however, is less intuitive.

As mentioned in the previous section, trim curve boundaries are usually irregular, resulting in difficulties in mathematically defining the new trimmed surface. If, however, the trim curve could be approximated in such a way that the resulting surface patches are still conforming to rectangular topology, so that they could be defined using tensor product polynomials, the difficulties associated with geometrical trimming could then be reduced significantly.

It is known that for a tensor product surface, the two-dimensional rectangular parametric space defining the surface also serves as the coordinate grid for the embedded curve. Each iso-parametric line of a surface patch is dependent on only one parametric variable (\( u \) or \( w \)) and independent of the second. These iso-parametric lines, in which each is a cubic B-spline curve by itself (for a bicubic B-spline surface), are obviously guaranteed
to be on the surface. This property of the iso-parametric lines consequently serves as the foundation for approximating the curve on the surface. The idea is presented below.

In the curve mapping process, after the curve has been mapped onto the surface by way of calculating the intersections of the projected lines from the curve with the surface, each point will produce an intersection Cartesian (xyz) coordinate as well as parametric (uvw) coordinate. Each parametric point has two independent iso-parametric lines associated with it. The idea is to join two parametric intersection points with their common iso-parametric line segment, if one exists, or to find an intermediate junction point to connect the two points with iso-parametric line segments. The result is a step-size approximation of the curve. These operations are done in the parametric space of the surface, which upon completion, is being mapped onto the modeling or surface space. However, joining each successive point in this manner is overkill. Thus certain approximation criteria have to be established. This is done by performing a simple linear curve-fit through a few points that has a deviation within a set tolerance. The two furthest points that produce a fit within the tolerance are then joined with their common iso-parametric line segments. Better approximation can conceivably be done by examining the curvature of the mapped curve. The step-size approximation in the parametric space is presented in Figure 10.

The resulting surface, with its step-size approximation trim curve boundary, can be represented by tensor product patches. The original patches that are traversed by the trim curve can be subdivided into tensor product subpatches.
Figure 10  Approximating a Trim Curve Using Surface Iso-parametric Line Segments.
7.0 Curve Mapping

The procedure taken in the curve mapping process as implemented in the ACSYNT B-Spline Module is presented here. The process involves the following steps:

1. The breakdown of ACSYNT surface models from component level to patch level for intersection, mapping and display.

2. The creation of B-spline mapping curves using Fleming's constraint-based inversion routine.

3. The interactive user determination of a reference mapped point on the surface patch.

4. The calculation of parametric tangent and normal vectors on the patch for mapping.

5. The transformation of the mapping curve to the new coordinate system with the user-defined surface point as the origin of the new system.
6. Refining criteria used in determining the distance of successive points on the mapping curve to be taken for projection using continuity information.

7. Interpolation approximation for getting intermediate mapped points.

8. The determination of the appropriate intersection data should multiple intersections be found.

9. Determination of criteria used in the approximation of boundary or edge mapped points and the determination of the appropriate surface boundaries and corners for out-of-bounds mapped curves.

The first eight procedures are discussed in detail in this chapter, while the last is presented in Chapter 8.
7.1 Surface Models at Patch Level

B-spline surfaces are inherently composite in nature. While a surface is composed of piecewise polynomial patches, the cubic basis functions defining the formulation ensure $C^2$ continuity across boundaries. Thus defining and displaying a B-spline surface does not need to be done patchwise. The ACSYNT B-Spline Module utilizes a linked-list format using the C programming language for storing all component data in a centralized format [Jone91]. A component is essentially a surface describing an aircraft part, such as the fuselage, nose, mid-section, afterbody, wings, etc. A separate model data structure is utilized to serve as the entry position to the component data. Besides non-uniform B-spline data, a set of bicubic Hermite surface data is maintained in the data structure for Hermite surface geometry as well as conversion from Hermite to B-spline form. Additionally, necessary information for PHIGS display such as component color and structure identifiers are included too. The example model and component linked-list data structures are shown in Figure 11.

For the purpose of curve mapping, the component data structure is further broken down into a patch structure, which is another linked-list. This list is maintained for direct and easy access to patch data, which is needed for the user to define a reference mapped point (step 4 in section 7.3 below) as well as for the necessary added data in the component data structure for the entry position and is shown in Figure 12.
typedef struct compdata_type {
    int    comp_number;    /* component number */
    char  *comp_name[20];   /* component name */
    int    acs_id;        /* structure identifier */
    int    nubs_id;
    int    *hull_id;      /* control polygon structure id */
    int    fillet_id;
    int    open[2];      /* open flag: 1 closed 0 open */
    int    color;        /* component color */
    int    existance;    /* 1 exists 0 nonexists */
    int    nu;          /* rendering in u */
    int    nw;          /* rendering in w */
    int    acs_ncross;   /* number of cross sections */
    int    acs_npts;     /* number of pts per xsection */
    float  ***acs_pts;  /* pointer to component points */
    float  ***acs_utan; /* pointer to tangents in u direction */
    float  ***acs_wtan; /* pointer to tangents in w direction */
    int    nu_knots;     /* number of u knots */
    int    nw_knots;     /* number of w knots */
    float  *u_knot;      /* u knot array */
    float  *w_knot;      /* w knot array */
    float  ***hull;      /* control hull */
    struct compdata_type next; /* pointer to the next component */
} comp_data;

typedef struct {
    int    num_comp;    /* number of components in model */
    int    acs_root;   /* strid executing all other strids */
    int    nubs_root;  /* non-uniform B-spline root id */
    int    int_root;   /* intersection curve structure id */
    int    fillet_root; /* fillet root identifier */
    int    shade;      /* shading flag */
    int    gauss_k;
    int    mean_k;
    comp_data *comp;    /* pointer to beginning of linked-list */
} MODEL;

Figure 11 Component and Model Data Structures
typedef struct patchdata_type {
    int patch_strid;     /* patch structure identifier */
    int patch_number;   /* patch number */
    int patch_color;    /* patch_color */
    int marker_id;      /* marker structure identifier */
    double marker_u;    /* u marker point on patch */
    double marker_w;    /* w marker point on patch */
    double marker_xyz[3]; /* marker xyz point on patch */
    double normal[3];   /* normal vector on marker point */
    double u_tangent;   /* u tangent on marker point */
    double w_tangent;   /* w tangent on marker point */
    double middle_xyz[3]; /* xyz point in the middle */
    double corner_xyz[3]; /* xyz point at uknot[3], wknot[3] */
    double u4w3_xyz[3];  /* xyz point at uknot[4], wknot[3] */
    double u3w4_xyz[3];  /* xyz point at uknot[3], wknot[4] */
    double end_u_tan[3]; /* u tangent at end of patch */
    double end_w_tan[3]; /* w tangent at end of patch */
    float patch_uknots[8];  /* u knot sequence for patch */
    float patch_wknots[8];  /* w knot sequence for patch */
    float patch_hull[4][4][3]; /* 3-d array control hull */
    float bicubic_coef[3][4][4]; /* bicubic coefficients */
} struct patchdata_type next;     /* pointer to the next patch */
} PATCH_DATA;

typedef struct compdata_type {
    int num_patch;     /* total # of patches in component */
    int num_u_patch;   /* # of u patches in component */
    int num_w_patch;   /* # of w patches in component */
    int intersection_flag; /* 0 no intersection 1 yes */
    PATCH_DATA *patch;  /* pointer to beginning of patch list */
} comp_data;

Figure 12  Patch Data Structure and Added Entries in Component Data Structure.

Curve Mapping 50
A single cubic patch consists of a minimum of 16 control points, 8 knots in each parametric directions (1 knot for each control vertex and 2 additional knots at each end of the control polygon) [Yama88] [Jone91], and 48 coefficients (see equation 5.10). For convention purposes, the numbering scheme of the patches (i.e. the link) is 'u-major' in a tensor product surface as shown in Figure 13.

7.2 Creation of Mapping Curves

ACSYNT is a surface modeller. Currently there is no curve function built into the software. Consequently, a module to create a mapping curve is needed.

The problem of projecting a curve onto a surface can be considered to be the same as projecting successive points from the curve to the surface. Therefore, in the simplest case, the mapping curve can be generated by simply providing the coordinates of an arbitrary number of points defining it in its modeling coordinates. However, the disadvantages of such a curve creation process are obvious. Firstly, unless the curve figure is made up of simple segments such as straight lines, to accurately model a curvature-intensive figure requires a considerably large number of data points. Generating these points manually can be extremely tedious. Furthermore, if at some point in time during the design process it is decided that a few more points are needed in between some generated points, this data has to be manually interpolated and calculated, which is inefficient. Secondly, the curve created does not any mathematical
Figure 13  Surface Patches Numbering Scheme.
basis, which means there are no intrinsic characteristics, such as curvature, defining it. This is a major flaw in any design process. Finally, the process is highly non-intuitive and inaccurate, making it unfeasible to be implemented.

As discussed in section 5.4, Fleming has incorporated continuity constraints in the inversion matrix of cubic B-spline curves. Basic defining data (interpolated) points and the associated curvature conditions and constraints are all that is needed to find the corresponding control points and knot sequence of a B-spline curve. There are two main functions developed for this inversion routine. One is Model_NUBS3_Curve which inverts the matrix and displays the curve created using the PHIGS Plus B-spline curve function. The other is Create_NUBS3_Curve, which inverts the matrix, calculates the control points and knot sequence and does not display the curve. This is the function that was employed here. The display of the curve is done by a C language routine utilizing the standard PHIGS functions. A curve input data file has to be created by the user to be read in by a routine, stored in a linked-list curve data structure, and displayed. The information needed in the input file includes:

1. Curve input header.

2. Number of data points.

3. Name of the curve part.

4. An array of xyz coordinates of the data points.

5. The continuity of each data point.

6. The continuity constraints of the data points (2 constraints for $C^0$, 1 for $C^1$ and none for $C^2$).
7. Information 2 to 6 is repeated for successive curve parts making up the complete curve model.

The entries in the curve data structure are very similar to the previously described surface component structures. A curve model structure is maintained for the entry position to the curve linked-list. A sample curve created is shown in Figure 14. The curve model, which is the ACSYNT logo, is made up of five closed curves.

It is worth mentioning that since the mapped curves are to be used ultimately as trimming curves, the curves created have to be closed. Thus the first and last data points in the input file for each curve part have to be equal. If not, an error message will be given and the curve will not be displayed.

A sample input file is shown and discussed further in Appendix A.
Figure 14  ACSYNT Logo - A Mapping Curve Example.
7.3 Determination of Reference Mapped Point

In any mapping process, the components present an unlimited surface space for the curve to be projected on. A method for an interactive user-defined mapped reference point has thus been developed. This is done by fully utilizing the piecewise nature of B-spline surfaces. The steps for user interaction in defining a reference mapped point is as follows:

1. The component for the curve to be mapped on is selected.

2. The patch data structure is used to display the selected surface component using different colors for different patches (Figure 15).

3. A reference patch is then selected.

4. The selected patch is displayed together with the $u$ and $w$ coordinates and a polymarker serving as a movable reference mapped point. A set of software valuator buttons are displayed as well (Figure 16). Using the buttons, the user can move the marker anywhere around the patch to get a satisfactory reference mapped point.

Once the reference mapped point is set, a normal vector at the point is calculated to serve as the curve projection vector. The calculation is shown in the next section.
Figure 15  Selected Component Displayed as Individual Patches.
Figure 16  User Interface in Selecting Reference Mapped Point.
7.4 Tangent and Normal Vectors on a Surface Patch

At any point \( P(u,w) \) on a tensor product patch, there are two tangent vectors, each associated with a parametric value \( (u \text{ and } w) \). Each vector is obtained by holding one parameter constant and differentiating with respect to the other. These vectors are given by:

\[
P_u(u,w) = \frac{\partial P}{\partial u} = \frac{\partial x}{\partial u} \hat{i} + \frac{\partial y}{\partial u} \hat{j} + \frac{\partial z}{\partial u} \hat{k}
\]  
\( (7.1) \)

\( \text{such that } u_{\text{min}} \leq u \leq u_{\text{max}}, \ w_{\text{min}} \leq w \leq w_{\text{max}}, \ w = \text{constant} \)

and:

\[
P_w(u,w) = \frac{\partial P}{\partial w} = \frac{\partial x}{\partial w} \hat{i} + \frac{\partial y}{\partial w} \hat{j} + \frac{\partial z}{\partial w} \hat{k}
\]  
\( (7.2) \)

\( \text{such that } u_{\text{min}} \leq u \leq u_{\text{max}}, \ w_{\text{min}} \leq w \leq w_{\text{max}}, \ u = \text{constant} \)

The magnitudes and unit vectors of the tangent vectors are given by:

\[
|P_u| = \sqrt{\left( \frac{\partial x}{\partial u} \right)^2 + \left( \frac{\partial y}{\partial u} \right)^2 + \left( \frac{\partial z}{\partial u} \right)^2}
\]  
\( (7.3) \)

\[
|P_w| = \sqrt{\left( \frac{\partial x}{\partial w} \right)^2 + \left( \frac{\partial y}{\partial w} \right)^2 + \left( \frac{\partial z}{\partial w} \right)^2}
\]

and

\[
\vec{n}_u = \frac{P_u}{|P_u|}, \quad \hat{n}_u = \frac{P_u}{|P_u|}
\]

\( (7.4) \)
For a B-spline patch, each tangent vector is calculated by partially differentiating the corresponding blending function \( N_i^k(u) \) or \( N_j^l(w) \) with respect to each parameter \( (u \text{ or } w) \).

The unit normal vector \( \hat{n}(u,w) \) is simply the vector cross product of the two tangent vectors, and is thus perpendicular to both vectors.

\[
\hat{n}_{uv} = \frac{P_u \times P_v}{|P_u \times P_v|}
\]

(7.5)

Figure 17 shows both tangent and the normal vectors on a bicubic patch.

The order of the cross product in equation 7.5 \( (P_u \times P_v \text{ or } P_v \times P_u) \) determines the sense the normal vector is pointing. Thus in an open surface, unless there is a predetermined modeling direction in surface creation, the normal vector direction could be hard to anticipate, however, reversing the direction of the vector is a simple task and can be easily done interactively upon visual inspection.

This normal vector serves as the normal projection line for the mapping process.
Figure 17  The Normal and Tangent Vectors on a Bicubic Patch.
7.5 *Transformation of Mapping Curves*

Both the B-spline curve and surface models are defined and created in their respective modeling coordinate system (MCS). For the mapping process to be carried out, the plane curve has to be transformed to the reference mapped point found earlier in the world coordinate system (WCS). The transformation of the 3-D coordinate systems has three degrees-of-freedom (DOF), one for each axis. An adaptation of PHIGS implementation on pattern mapping [Gask92] is used. The first DOF is fixed by lining up the normal vector of the plane curve (z-axis) with the normal vector of the mapped point found earlier. The second DOF is anchored by aligning the x-axis of the curve with one of the parametric tangent vectors of the surface. The choice is strictly arbitrary, however, depending on the definition of the surface parameter directions, the curve might be mapped upside-down, so a function to interactively flip the curve around is necessary.

Fixing the third degree-of-freedom is more involved. It seems logical to go about doing it the same way as the previous two, i.e. lining up the y-axis of the curve with the second tangent vector of the mapped point. This is all right if the two tangent vectors are perpendicular to one another, which happens only on a perfectly rectangular patch. If they are not perpendicular, then the mapped curve will be skewed, essentially being mapped in a parallelogram manner. As a result, a third orthogonal axis on the mapped point has to be calculated. This is done by taking the cross product of the normal vector with one of the two tangent vectors chosen to fix the second DOF. The y-axis of the curve plane is then transformed to line up with this third orthogonal axis. Figure 18 shows the transformation idea.
Figure 18  Transformation of Plane Curve to Mapped Reference Point.
Deriving the transformation matrix is the heart of the procedure. Uicker et al. present comprehensive discussions on rotation and translation of spatial mechanisms [Uick65]. Notations used in this thesis follow that presented in Craig [Crai89].

For a general case, transforming a point \( P \) defined in coordinate system \( B \) to coordinate system \( A \) involving both rotation and translation in space (see Figure 19) is given by:

\[
^A\vec{P} = ^A{T}_B^B\vec{P}
\]  
(7.6)

where

\( ^A\vec{P} \) = Point \( P \) defined in coordinate system \( A \)

\( ^A{T}_B \) = Transformation matrix from system \( B \) to \( A \)

\( ^B\vec{P} \) = Point \( P \) defined in coordinate system \( B \)

The definition of \( ^A{T}_B \) is:

\[
^A{T}_B = \begin{bmatrix}
^A{R} & ^A{P}_{\text{orig}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]  
(7.7)

where

\( ^A{R} \) = Rotational matrix from system \( B \) to \( A \)

\( ^A{P}_{\text{orig}} \) = Origin of system \( B \) expressed in system \( A \)

An important characteristic of the rotational matrix \( ^A{R} \) is such that each column represents an orthogonal unit vector, which is equivalent in saying that the first column is the projection of \( x_B \) on \( x_A \), \( y_A \), and \( z_A \), the second column is the projection of \( y_B \) on \( x_A \), \( y_A \), and \( z_A \), and the third column is the projection of \( z_B \) on \( x_A \), \( y_A \), and \( z_A \). Or:

\[
^A{R} = \begin{bmatrix}
\hat{x}_B \cdot \hat{x}_A & \hat{y}_B \cdot \hat{x}_A & \hat{z}_B \cdot \hat{x}_A \\
\hat{x}_B \cdot \hat{y}_A & \hat{y}_B \cdot \hat{y}_A & \hat{z}_B \cdot \hat{y}_A \\
\hat{x}_B \cdot \hat{z}_A & \hat{y}_B \cdot \hat{z}_A & \hat{z}_B \cdot \hat{z}_A
\end{bmatrix}
\]  
(7.8)

Thus, for curve mapping process, the first and second columns of the rotation matrix are the normalized (unit) vector of the third orthogonal axis and one of the tangent vectors on

Curve Mapping 64
Figure 19  General Spatial Transformation.
the mapped point respectively, and the third column is the unit vector of the mapped point normal vector. The positional matrix (fourth column of the transformation matrix) is then the location of the mapped point origin with respect to the absolute origin.

7.6 Determination of the Distance between Successive Mapped Points

The amount of time required to project a curve onto a surface is directly proportional to the complexity of the curve, and to a certain extent, the surface. Cutting down on the number of points needed to be projected is an important aspect in minimizing the processing time. A simple method has been developed to do just that.

In the curve creation process described above (section 7.2), continuity information for each data point is furnished. It is obvious that two \( C^0 \) continuity data points make up a straight line. It is thus unnecessary to generate intermediate points for projection since they would all be lying on the same line. Projecting the two \( C^0 \) points and using the linear interpolation technique described in the next section to obtain the intermediate points to fit through a curve is much faster than calculating intersections. However, two important conditions must be satisfied before this approach can be taken. First, the curve has to be lying entirely within the boundary of the surface. In another words, it should not go out-of-bounds of the surface boundary. Secondly, the patch information, specifically,
the patch numbers where the intermediate points lie on have to be known for interpolation to be carried out. This will be discussed in the next section.

Better criteria for determining the distance between curve points to be taken for projection can conceivably be done more robustly by examining the piecewise curvature of the mapping curve. Currently, all segments which contain either $C^1$ or $C^2$ continuous data points are broken down to get ten intermediate points for projection.

7.7 Interpolation Approximation

The linear interpolation scheme to generate intermediate points between two $C^0$ data points is done in the parametric space. In this space, the two parameters ($u$ and $w$) for each of the two points are found from the intersection routine. Depending on the number of interpolated points needed in between them (hard-coded to be ten), the difference between one of the two parameters (normally the greater difference of the two) is divided by this number of points (ten), thereby fixing one unknown parameter. The second unknown parameter is then calculated by the equation:

$$u = \frac{(w - w_a)(\mu_b - \mu_a)}{(w_b - w_a)} + \mu_a$$  \hspace{1cm} (7.9)
where

\[ u = \text{unknown parameter to be found} \]
\[ w = \text{fixed interpolated parameter} \]
\[ u_a, w_a = \text{known parameters of the first point} \]
\[ u_b, w_b = \text{known parameters of the second point} \]

The reason that the patch information has to be known for interpolation calculation is because if the intersection routine normalizes all patch knots while finding intersections, the intersected parameters found would be returned as normalized parameters too. These normalized parameters are meaningless without the knowledge of their original parametrization. Nonetheless, if the patch number is known, normalized parameters can then be transformed back to their original parameters. Moreover, it is also important to know that interpolation can be performed on the surfaces since chord length parametrization is used in creating the surfaces. With this parametrization, the knot spacing is proportional to the distance between the data points.

### 7.8 Processing of Multiple Intersection Points

In the projection process, projecting a curve point onto the surface patches occasionally produces multiple intersections. This happens when the surface folds backwards (e.g. a cylinder) or when the projection line cuts the surface at high curvature areas. This multiple intersection problem arises both in a single patch or different patches on the same surface, as shown in Figure 20. The top figure shows a projection line intersecting
a single patch twice and the bottom figure shows the line intersecting at three different patches. In either case, only one intersection is needed, namely the one closest to the projection point, or curve. The determination of this correct point is simple enough in that only a projection line that resulted in multiple intersection points has to be processed. Distances from the projected point to each intersection are calculated and compared. The one with the shortest distance is kept and the rest discarded. However, manipulation of the linked-list data structure of successive mapped points has to be done. This includes an insertion and deletion algorithms.

The important steps in the curve mapping process have been discussed in detail in this chapter. The last designing problem involves finding the correct surface boundaries and corners to close the mapped curve if it is mapped out-of-bounds. This issue will be discussed in the next chapter.
Multiple Intersections with a Patch

Intersection points on a single patch

Multiple Intersections with a Surface

Intersection points on three different patches

Figure 20  Multiple Intersections Between a Projection Line and a Surface/Patch
8.0 Boundary Information Processing

8.1 Introduction

Mapping a curve onto a surface will invariably produce some special cases. Some of these cases have been described in the previous chapter, such as the orientation of the curve relative to the surface and the processing and determination of the correct intersection point should multiple intersections be found. One other important case that will inevitably arise in the mapping process is when the curve is mapped out-of-bounds on the surface. Several crucial issues will be encountered when this condition happens. These are all associated with the need to determine surface edge or boundary information, because the mapped curve should not be left open on the surface, meaning that somehow the mapped curve has to be closed. Closed curves are needed to define a trimming loop on a surface. They are also needed for shading.
Figure 21 shows a few examples of out-of-bounds mapped curves. It is apparent that the number of cases that will appear are unpredictable. For instance, Figures 21a and 21b produce the same mapped shape on the surface, however, determining which boundaries and corners of the surface to close up the curve is important for the mapping process to be correct.

This chapter will explore the several ways in processing the surface corner and boundary information for out-of-bounds curves. Recognizing that the cases involved are numerous, only a few cases will be solved. A definite pattern on how other cases will occur could be determined to make the program more robust.
Figure 21  Example Cases of Out-of-Bounds Mapped Curves.
8.2 Determination of Edge Points

As discussed before, the mapping process is accomplished by producing a number of points on the curve and perpendicularly projecting them in successive order onto the surface. The projection operation is the same as performing a line-surface intersection algorithm. This intersection method provides both advantages and disadvantages in the case of determining an edge point.

Successively projecting points in an orderly manner onto a surface to find intersections provides a means of determining the need to find an edge point. This is done by keeping track of the status of each intersection operation. Any change in the intersection status of two projection points signifies the point where the mapped curve is going out of (or into) the surface boundary and thus an edge point needs to be found. For instance, if a current point is found not to have an intersection with the surface while the previous point has, it shows that the curve is going out of the surface boundary between the two points, and a point on the boundary of the surface has to be found. Likewise, if a current point is found to have an intersection with the surface while the previous point did not, the curve is going into the surface in between the two points and again an edge point needs to be found. This information is achieved by simply keeping track of the intersection operation.

On the other hand, finding an edge point is a problem not easily solved. One way of doing this is to subdivide the B-spline line segment between the two points where an edge point is known to lie between into more points and utilize the brute-force method to project these new points onto the surface until one is found to be within the set tolerance close enough to the boundary of the surface. As accurate as it might be, this method is
inefficient. The time needed in subdiving the B-spline line segments and then projecting them to find intersections and checking for tolerances will consume too much processing time, especially when the number of points are many.

An approximation criterion has thus been developed to solve this problem. The method employed is again based on the tensor-product characteristics of a bicubic surface. There are two assumptions in the algorithm:

1. The surface has to be open at the sides where the curve is going out of the boundaries.

2. The successive projection points must be close enough so that the one right before going out of the boundary or the first to come into the boundary has to be intersecting with one of the perimeter patches. Perimeter patches are those that are along the perimeter of the surface as shown in Figure 22a.

With the second assumption, the algorithm narrows its searching compound to only the minimum number of patches of interest and simplifies the search process and determination criterion. The perimeter patches can again be categorized into corner and side patches as shown in Figure 22b. The algorithm approximates the parameter of the edge point using the constant parameter of the isoparametric line that the point lies on. If the point lies on one of the side patches, the approximation criterion is simpler since there is only one parametric surface boundary on its sides. Corner patches need further processing to determine which one of the two surface boundaries is the correct one.

The parameter on the surface boundary of a side patch can be one of the four following:

1. The first knot of $u$ parameter;

2. The last knot of $u$ parameter;
Figure 22  Surface Perimeter Patches.
3. The first knot of \( w \) parameter;
4. The last knot of \( w \) parameter.

It should be cautioned, however, that the knot sequence of each parametric direction has three extra knots at both ends (bicubic patches). So the 'first' knot stated above is essentially the 'fourth' in the knot sequence. Similarly, the 'last' knot is actually the 'fourth last' in the knot sequence. For example, if the knot sequence in the \( u \) direction of the surface looks like:

\[-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\]

Then the two boundary knots are 0 as the 'first' and 6 as the 'last' in accordance with the convention here. This leads to the method for arriving at the approximated edge point. If the point lies on one of the side patches where the surface \( u \) parametric boundary is the first knot, then the \( u \) parameter of the edge point is taken to be the first \( u \) knot, and its \( w \) parameter is kept as that of the intersection point. This analogy applies to all other side patches. For instance, if the intersection point lies on one of the side patches where the surface \( w \) parametric boundary is the last knot, then the \( w \) parameter of the edge point is this last \( w \) knot of the surface, and the \( u \) parameter is that of the intersection point. This is shown graphically in Figure 23.

If the perimeter patch is one of the four corner patches, further processing has to be done. This is because there are two surface boundaries associated with each corner patch. This means that there are two possibilities for the edge point to lie on. A distance check has to be done to determine the solution. This is done by calculating and comparing the distances of the intersection point to the two edges. The closer of the two will be taken as the edge parameter. This is shown graphically in Figure 24.
Figure 23  Determination of Edge Parameter for a Side Patch.

Boundary Information Processing
Figure 24  Determination of Edge Parameter for a Corner Patch.

Parametric values of edge point:
\[ u = 5.5 \]
\[ w = 10 \]
8.3 Determination of Edge Boundaries

After the edge points are found, they have to be connected along the surface boundaries to close the mapped curve. This introduces some new problems because of various possible cases as shown in Figure 21 earlier. Specifically, referring to two map cases in the figure (a and b), the boundaries to be connected can be totally different even though the partially mapped curves are identical in both cases. This depends on the original shape and position of the curve model. The next two map cases (c and d) show a similar problem. Even though the two edge points are on the same surface boundaries, the task of joining them is far from simple. The problem gets more complicated when the mapped curve goes out of and into the boundaries of the surface more than twice, as shown in Figures 21e and f.

An innovative way of determining the correct edge boundaries has been implemented for the case where only two edge points are encountered. One is when the curve is going out of the surface boundary and the other as the curve is coming into the boundary.

The crucial point in determining which boundary or boundaries are needed to close the mapped curve is to find out which are the corners of the surface that need to be included. This can be done by projecting each successive corner along the opposite direction of curve projection line back into space and into the plane defining the curve (Figure 25). The intersection point of this corner projection line with the plane is then used to test if it lies internal or external to the curve. Any one corner which is internal to the curve signifies the two boundaries associated with it have to be part of the mapped curve. The two edge points of the mapped curve are then joined appropriately.
The internal and external test is done by finding the number of times an infinitely extended line from the test point crosses the curve. The specific algorithm is described below.

The plane curve is first divided into line segments in its own $xy$ modeling coordinates. The $x$ coordinate of the test point, which has also been transformed to the curve modeling coordinates, is tested with each successive curve line segments to see if it lies in between the segment. If it does, then the point is projected in positive $y$ direction to see if it intersects with this particular curve line segment. Any intersection is kept track of by a counter. The process is repeated with all line segments. Finally, the point can be determined if it lies internal or external to the curve from the counter value. If the value is odd, which means that the point passes the curve an odd number of times, then the point is internal, otherwise, it is external to the curve. The idea is shown in Figure 26.
Mapping curve on a 2-D plane

--- → Curve Projection Line

← Reverse Corner Projection Lines

Figure 25  Reverse Projection of Surface Corners to the Plane Curve.
○ Internal points  × Crossings  → Positive y infinite lines
All point extensions cross figure odd number of times
(a) Internal Points

○ External points  × Crossings  → Positive y infinite lines
All point extensions cross figure even number of times
(b) External Points

Figure 26  Internal-External Test.

Boundary Information Processing 83
9.0 Results

This chapter presents several photographic results of the completed curve mapping routine as implemented in the ACSYNT B-Spline Module. All images are generated on an IBM RS/6000 workstation.

Figure 27 shows a B-spline ACSYNT logo curve model generated using five closed curves. This geometry was generated using the previously discussed constraint-based inversion technique. Figure 28 shows a mapped ACSYNT logo on an aircraft fuselage. Since the fuselage is cylindrical in shape, the mapping or projection process produces multiple intersection points from each projection point. This example demonstrates the need to process such a multiple intersection problem. Figure 29 shows a mapped model in four views. It is interesting to note that all mapped points lie exactly on the surface following the surface curvature. Figures 30 and 31 show the clipped mapped models, in which both curves were mapped out of the boundaries of the surface. While producing the same mapped shape on the surface, the resulting boundaries being joined to close the curves are different. Figure 32 shows multiple mapped curves on a surface model.
Figure 27  An ACSYNT B-Spline Logo.
Figure 28  An ACSYNT logo mapped onto the Fuselage of an Aircraft Model.
Figure 29  A Mapped Model in Four Views.
Figure 30  An Out-of-Bounds Curve Covering One Surface Corner.
Figure 31  An Out-of-Bounds Curve Covering Three Surface Corners.
Figure 32  Multiple Mapped Curves on a Surface Model.
10.0 Conclusion

This research discusses the mapping and projection of curves onto B-spline surfaces. The mapping process utilizes normal projection of points on a plane curve onto the surface. A B-spline curve generation routine has been included. Implementation has been completed in the ACSYNT B-Spline Module. All necessary user interactivity in the mapping process has been coded. Several problems associated with the mapping process have also been discussed and implemented, such as the process of multiple intersection points and mapping out-of-bounds curves. The usefulness of this mapping process has also been discussed. This includes adding an aesthetic element and added detail to a surface model, visually determining exterior physical details on a design surface or model, and more importantly, treating the curves as trim curves. The mapped curve as it is can be used for visual trim curves since it is guaranteed to be closed, even if the curve is mapped out of the surface boundaries. A new approach in approximating the mapped curve using the existing iso-parametric line segments on the surface which produces surface patches that can be subdivided into tensor product patches might possibly lead to the development of geometrical trimming algorithms.
The algorithm implemented in this research is not robust enough to solve all possible mapping cases. Some of them will be discussed below.

1. Though simple criteria (e.g., no points are needed to be projected in between two \( C^0 \) continuous data points) have been incorporated in generating points on the curve for speedy mapping, the method can be improved by studying the curvature information of the curve more closely. For instance, at portions where curvature is higher, more points should be generated. This is important not only for mapping purposes but also when approximating the mapped curve using isoparametric line segments of the surface.

2. Possible cases encountered when mapped curves are out of the boundaries of the surface are numerous. A robust enough algorithm that can find all necessary corners and boundaries of the surface to join and close the curves no matter how many times the curve is going out of and coming into the surface needs to be developed.

3. Out-of-bounds curves will not only be encountered in open surfaces, but in closed surfaces or surfaces that curve back as well. An example is shown in Figure 33. This is a more complex problem as the edge boundary where the curve goes out-of-bounds is not necessarily a surface boundary. This would be an interesting problem to be tackled.

After the curve model has been mapped onto the surface, it would also be necessary to process further, such as to show the new model as a shaded image. It should be kept in mind that the mapped curve is a 3-D space curve, which is not possible to perform any shading on. However, the surface underlying the curve is still a surface representation,
thus it should be interesting to know that it might be possible to perform a shading operation on the same surface model twice, each time on a separate portion of the surface; the mapped portion and the remaining portion. This is especially needed for displaying exterior details.
Figure 33 Curve Mapped Out-of-bounds on a Curved Boundary of a Surface.
11.0 References


References


References


Also submitted to Computer Aided Geometric Design.


Appendix A  Curve Mapping Module User Guide

A.1 Introduction

The user guide consists of a simple step-by-step tutorial on the use of the module; from the reading in of a default data file to the actual mapping process, as well as a more detailed reference on each function. The tutorial is meant to serve as a quick overview of the capabilities of the functions. The module reference acts as a detailed functional description compiled in the order of menu layer traversal.
A.2 Tutorial

The tutorial provides a step-by-step procedure in providing a quick overview of the defining features of the curve mapping routine. The user is assumed to be familiar with the basic operation of ACSYNT, such as the menu structure, software valuator buttons, and the overall screen layout. The input data files for both the surface and curve models are provided as examples. No knowledge of B-splines is needed in completing the task.

Since the module was developed on ACSYNT B-Spline Module version 1.2.2, the tutorial steps will be based on this version of the software too. However, the steps in the mapping process should not have any discrepancies if the module is compiled in the integrated ACSYNT (version 2.0 with SURFACE module representing the B-Spline Module) besides the direct reading in of a B-spline surface data file; which is not available in ACSYNT 2.0.

Loading B-Spline Surface Model:

1. Choose menu sequence

   FILE - READ NURBS - SHOWTIME

2. Type in the filename

   wave.surf
and hit <ENTER>. A B-spline wavy surface will be displayed in the geometry window.\(^1\)

**Loading B-Spline Curve Model (Mapping Curve):**

1. Choose menu sequence

   \textit{RETURN - GEOMETRY - B-SPLINE - CURVE MAP - LOAD CURVE}

2. Type in the filename

   `acslogo.crv`

   and hit <ENTER>. A B-spline ACSYNT logo made up of five closed segments will be displayed in the geometry window. This is the curve model to be mapped onto the loaded surface.

**Defining Mapped Reference Point:**

1. Choose menu item \textit{DEFINE POINT}

2. Select the surface component.

   After the selection, the surface will be re-drawn as individual patches with different colors for each patch.

3. Select the middle white patch.

   Upon the selection, the surface component will be re-drawn again, this time back to its original color with the selected patch highlighted using a different color.

---

\(^1\)The integrated ACSYNT (version 2.0) does not allow reading in of a B-spline file. Instead, B-spline geometry will be automatically created from the Hermite file read in.
4. Choose menu item **ACCEPT**
   This selection will prompt the program to show the selected patch on its own with a polynumber drawn at one corner (at lower knot values of both \( u \) and \( w \) knots), and axis legends showing the directions of \( u \) and \( w \) parameters in the geometry window. The menu window will display 10 buttons, 9 of them with directional arrows, which are the software valuators buttons to move the pointer (polynumber) on the patch around and the lowest button with the text 'DONE'.
   The 9 arrow buttons move the pointer on the patch according to a specified pattern, which will be described in the Reference section.

5. Select the middle pointer valuator button (with a square drawn in the middle).
   This will move the pointer to the middle of the patch. This pointer position is the reference mapped point.

6. Select the bottom 'DONE' button to anchor the pointer position.
   The geometry window now displays some additional figures. Firstly, three orthogonal lines representing the mapped coordinate system are displayed. The green line is the tangent direction of the \( u \) parameter at the mapped point, the purple line is the tangent direction of the \( w \) parameter, and the white line is the normal to the patch at the reference mapped point. This normal line is the projection line. Secondly, the curve model (ACSYNT logo) has been transformed to be perpendicular to the normal projection line and displayed above the patch. This is the position that is going to be perpendicularly mapped onto the surface.

**Rotating Curve:**

- Choose menu item **ROTATE CURVE** twice.
Each time the menu is chosen, the curve is rotated 90 degrees counter-clockwise.

**Mapping Curve:**

- Choose menu item **MAP CURVE**
  
The curve is now mapped onto the surface.

**Mapping Multiple Curves:**

More than one curve model can be mapped onto the surface and shown simultaneously. The following are the steps to perform multiple mapping.

1. Choose menu sequence

   **RETURN - CURVE MAP - LOAD CURVE**

2. Type in the filename

   `circle.crv`

   and hit <ENTER>. A B-spline circle will be displayed in the geometry window.

3. Select the first patch (red patch at the top) as the reference patch.

4. Do not move the patch pointer. Select the 'DONE' button.

5. Select menu item **MAP CURVE**

   The circle mapped is out-of-bounds. Notice that the appropriate edges and corner of the surface are joined to the mapped arc to close the mapped curve.

6. Select menu item **DISPLAY ALL**

   Both the mapped ACSYNT logo and the circle are shown on the surface.
A.3 Reference

This reference section provides detailed descriptions of each menu option in the Curve Mapping Module. Familiarity of the ACSYNT menu structure and screen layout is assumed, though no knowledge of B-splines is required.

All menu items described below are reached via the menu path

FILE - READ NURBS - SHOWTIME - GEOMETRY - B-SPLINE - CURVE MAP

A.3.1 Load Curve Menu (Curve Data File Creation)

Upon the selection of this menu item a curve data filename has to be entered for the program to read and store the data in CURVE_MODEL and CURVE_DATA data structures.

The creation of a curve data file must adhere to a prescribed format for the program to read properly. A sample data file that consists of two closed curves with explanatory line numbers is shown in Figure 34. The line-by-line explanation is given in Table 1.

The corresponding curves created are shown in Figure 35. It should be noted that although two curves are created, both are collectively regarded as one curve model, as they are created in one single data file. Each curve model will be mapped onto the surface in one mapping operation.
Data file: 
CURVE INPUT FILE
5
RANDOM
Data-Points:
-2.00 -2.00 0.00
-2.00 2.00 0.00
  2.00 2.00 0.00
  2.00 -2.00 0.00
-2.00 -2.00 0.00
Continuities:
  2
  0
  1
  2
  2
Constraints:
  0.00 0.00 0.00
  0.00 0.00 0.00
  1.00 0.00 0.00
5
CIRCLE
Data-Points:
-4.00 -4.00 0.00
-4.00 4.00 0.00
  4.00 4.00 0.00
  4.00 -4.00 0.00
-4.00 -4.00 0.00
Continuities:
  2
  2
  2
  2
Constraints:
  None

Figure 34 Sample Curve Data File.
<table>
<thead>
<tr>
<th>Line Number</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Title of data file (verbatim).</td>
</tr>
<tr>
<td>2</td>
<td>Number of data points (total number + 1 (last = first) to close the curve).</td>
</tr>
<tr>
<td>3</td>
<td>Name of curve part (arbitrary).</td>
</tr>
<tr>
<td>4</td>
<td>Heading of data points (verbatim).</td>
</tr>
<tr>
<td>5 - 9</td>
<td><em>xyz</em> coordinates of each data point (recommended that point (0,0,0) be at the middle of entire curve model). The last point must be equal to the first to close the curve.(^2)</td>
</tr>
<tr>
<td>10</td>
<td>Heading of continuity of each data point (verbatim).</td>
</tr>
<tr>
<td>11 - 15</td>
<td>Continuity of each data point ($C^0 - 0$, $C^1 - 1$, $C^2 - 2$).</td>
</tr>
<tr>
<td>16</td>
<td>Heading of continuity constraints (verbatim).</td>
</tr>
<tr>
<td>17 - 18</td>
<td>The two tangent constraints for the second data point ($C^0$ continuity).(^3)</td>
</tr>
<tr>
<td>19</td>
<td>The one tangent constraint for the third data point ($C^1$ continuity).</td>
</tr>
<tr>
<td>20 - 35</td>
<td>The second curve part, following the previous rules and order.</td>
</tr>
<tr>
<td>35</td>
<td>The word 'None' for no continuity constraints (all points are $C^2$ continuous).</td>
</tr>
</tbody>
</table>

It is interesting to note that although both curves are created with rectangularly defined data points, with the different continuities on the data points, the shapes of the curves are totally different.

---

\(^2\) An error message will be generated by the program if the last point is not equal to the first.

\(^3\) Each $C^0$ continuity has 2 tangent constraints, each $C^1$ has 1, and $C^2$ has none.
Figure 35  Curves Created Using Data File Shown in Figure 34

Appendix A  Curve Mapping Module User Guide  109
A.3.2 Define Point Menu

Upon the selection of this menu item, the following message will appear on the message window:

SELECT COMPONENT FOR MAPPING

A component from the surface model has to be selected. This selected component is the one for the curve model to be mapped on. After selecting the component, it will be shown alone with all its defining patches drawn in different colors. A message will appear as follows:

SELECT REFERENCE PATCH TO MAP

A patch is to be selected from the component. This patch will serve as a reference patch for the mapping process. After a patch has been selected, the component will be redrawn as its original color, with the selected patch highlighted in another color. A message will appear:

ACCEPT OR REJECT PATCH
A.3.2.1 Accept Menu

This menu selection accepts the reference patch selected. Several screen changes happen after the selection.

1. The selected patch is drawn with the axis legends indicating the $u$ and $w$ parametric directions. Also in the geometry view is a little circular polymarker acting as a pointer on the patch for the reference mapped point. This marker is by default set at the lower parametric directions of each patch (uknot[3] and wknot[3]).

2. A second set of menus is shown. Also in the menu view is a new set of ten buttons. Nine of them have arrows drawn in them and the last and bottom-most rectangular button has a text written in as 'DONE'.

The nine arrow headed buttons are the software (pointer) valuator buttons. Each arrow indicates a corresponding parametric direction of the patch pointer movement. The directions of movement are as shown in Table 2 on the next page:
<table>
<thead>
<tr>
<th>Button:</th>
<th>Direction:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top left (North West)</td>
<td>decrement in $u$, increment in $w$.</td>
</tr>
<tr>
<td>Top center (North)</td>
<td>constant in $u$, increment in $w$.</td>
</tr>
<tr>
<td>Top right (North East)</td>
<td>increment in $u$, increment in $w$.</td>
</tr>
<tr>
<td>Middle left (West)</td>
<td>decrement in $u$, constant in $w$.</td>
</tr>
<tr>
<td>Middle (Center)</td>
<td>pointer moves to the middle of patch</td>
</tr>
<tr>
<td>Middle right (East)</td>
<td>increment in $u$, constant in $w$.</td>
</tr>
<tr>
<td>Bottom left (South West)</td>
<td>decrement in $u$, decrement in $w$.</td>
</tr>
<tr>
<td>Bottom center (South)</td>
<td>constant in $u$, decrement in $w$.</td>
</tr>
<tr>
<td>Bottom right (South East)</td>
<td>increment in $u$, decrement in $w$.</td>
</tr>
</tbody>
</table>

After the desired pointer location is reached, the **DONE** button should be selected.

With the selection of the **DONE** button, the software valuator buttons will disappear. In the geometry view, the picked patch will be shown together with three orthogonal axes serving as the coordinate system of the reference mapped point. The green line is the tangent line of the $u$ parameter at the point, the purple line is the tangent of the $w$ parameter at the reference point, and the white line is the normal. This normal line is the projection line. The curve model is also displayed in the view, slightly offset from the patch. A black-and-white printed figure of how the screen will look like at this point is shown in Figure 36 on the next page.
<table>
<thead>
<tr>
<th>CURVE MAP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>DEFINE POINT</td>
<td></td>
</tr>
<tr>
<td>FLIP CURVE</td>
<td></td>
</tr>
<tr>
<td>ROTATE CURVE</td>
<td></td>
</tr>
<tr>
<td>MAP PREVIEW</td>
<td></td>
</tr>
<tr>
<td>MAP CURVE</td>
<td></td>
</tr>
<tr>
<td>DISPLAY ALL</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 36**  Screen Layout Showing Picked Patch, Mapping Coordinates and Mapping Curve.
A.3.2.2 Reject Menu

This menu item should be selected if the selected reference patch is to be discarded and re-selection is desired. Upon its selection, the message window will re-display the following message:

SELECT REFERENCE PATCH TO MAP

Procedure shown in section A.3.2 will be repeated as long as this item is selected until 
ACCEPT menu is selected.

A.3.3 Flip Curve Menu

This menu item is used to flip the curve around to the other side of the surface component. The normal direction of the reference point is reversed to achieve this operation. The original and flipped curve model are shown in Figure 37.
Figure 37  Original and Flipped Curve Model.
A.3.4 Rotate Curve Menu

This menu item is for rotating the curve. Rotation is set at 90 degrees counter-clockwise. The rotation is achieved by manipulating the direction of the $u$ and/or $w$ tangent vectors.

Figure 38 shows successive rotation of a curve model from its original position.

A.3.5 Map Preview Menu

Upon selection of this menu item, projection lines will be drawn from the curve to the surface, previewing how the curve will be mapped onto the surface. Figure 39 shows the map preview in four views.

A.3.6 Map Curve Menu

This is the actual menu item for mapping. The time needed in the mapping process depends on several factors, such as the complexity of the curve, the nature of surface curvature (for instance, if the surface curls back, multiple intersection check and elimination procedure has to be performed), the bounding of the curve inside surface
boundary (determines if edge and corner information has to be processed as described in Chapter 8), as well as the load of the machine processor at that particular time of execution.

During the calculation process, a message will appear as follows:

CALCULATING INTERSECTIONS, PLEASE WAIT....

After all mapped points have been found, another message will appear:

DRAWING MAPPED CURVE....

After which the mapped figure will be drawn together with the entire surface model.

A.3.7 Display All Menu

If multiple curves are mapped onto the surface models, upon the selection of this menu item will display all the mapped figures. The curve models could be the same or different, as are the mapped components.
Figure 38  Rotation of Curve Model.
Figure 39  Map Preview Shown in Four Views.
Appendix B  C Source Code Functional Description

B.1 Introduction

All routines in curve mapping module were coded in ANSI C programming language as part of the ACSYNT B-Spline Module. Invariably, some routines, such as menu interface and surface creation functions are called from the originally developed code. This appendix includes only functional descriptions of routines developed for the curve mapping module.

The routines are sorted by alphabetical order of the C source code files that they are included in.
B.2 bslnsurf.c

B.2.1 bsLnSurfInt

Description:

Finds the intersection points between the line determined by the points pt1 and pt2 and the surface determined by C.

Function Prototype:

```c
int bsLnSurfInt (FLOAT pt1[3], FLOAT pt2[3], FLOAT C[3][4][4], FLOAT TOL,
                 FLOAT PARAMS[MAXIPTS+1][3], FLOAT IPT[MAXIPTS+1][3], int *NUM);^1
```

Input Arguments:

- `FLOAT pt1[3]` the two points defining the line
- `FLOAT pt2[3]` bicubic coefficients defining the surface
- `FLOAT C[3][4][4]` tolerance to determine if two points are the same
- `FLOAT TOL`

Output Arguments:

- `FLOAT PARAMS[MAXIPTS+1][3]` parametric values of intersection point
- `FLOAT IPT[MAXIPTS+1][3]` spatial values of intersection point
- `int *NUM` number of intersection points

Function Output:

```c
int
```

1 - successful, 0 - unsuccessful

Author:

Dr. C. Feustel

---

^1"FLOAT" is a user-defined type as defined in file rtype.h. It can be of type float or double.
B.3 curve_map_menu.c

B.3.1 load_curve_menu

Description:

Menu module for curve loading (reading).

Function Prototype:

    void load_curve_menu (MODEL *);

Input Arguments:

    MODEL *Model         pointer to MODEL data structure

Output Arguments:

    None

Function Output:

    None

B.3.2 map_curve_menu

Description:

Menu module for curve mapping.

Function Prototype:

    void map_curve_menu (MODEL *, CURVE_MODEL *, MAPPED_CURVE *[]);
Input Arguments:

MODEL *Model
CURVE_MODEL *Curve_Model
MAPPED_CURVE *Mapped_Curve[]

pointer to MODEL data structure
pointer to CURVE_MODEL data structure
pointer to MAPPED_CURVE data structure

Output Arguments:

MAPPED_CURVE *Mapped_Curve[]

pointer to MAPPED_CURVE data structure
(passed back from draw_mapped_curve via get_map_data)

Function Output:

None

B.3.3 confirm_menu

Description:

Menu module for the status of the acceptance of the picked patch.

Function Prototype:

int confirm_menu (void);

Input Arguments:

None

Output Arguments:

None

Function Output:

int confirmation
0 - accept
1 - reject

Appendix B  C Source Code Functional Description 123
B.4 define_point.c

B.4.1 define_point

Description:

To identify the patch that has been picked by the user for the curve to be mapped on.

Function Prototype:

    void define_point (MODEL *, comp_data **, PATCH_DATA **, CURVE_MODEL *);

Input Arguments:

    MODEL *Model
    CURVE_MODEL *Curve_Model

        pointer to MODEL data structure
        pointer to CURVE_MODEL data structure

Output Arguments:

    comp_data **ptrptr_to_comp
    PATCH_DATA **ptrptr_to_patch

        double pointer to comp_data data structure
        double pointer to PATCH_DATA data structure

Function Output:

None
B.4.2 get_patch_picked

Description:

To get the patch of the component picked by the user.

Function Prototype:

PATCH_DATA *get_patch_picked (comp_data *);

Input Arguments:

comp_data *pr_to_comp  
pointer to data structure of component to be mapped on

Output Arguments:

None

Function Output:

PATCH_DATA *ptr_to_patch  
data structure of patch picked
B.5 draw_bspline_comp.c

B.5.1 draw_bspline_comp

Description:

Module to draw a B-spline component.

Function Prototype:

```
void draw_bspline_comp (comp_data *, int, PATCH_DATA *);
```

Input Arguments:

- comp_data *comp
- int color_on_off
- PATCH_DATA *picked_patch

  pointer to comp_data data structure
  color of patch toggle
  pointer to the picked patch

Output Arguments:

None

Function Output:

None
B.5.2 display_colorful_comp

**Description:**

To re-display the component for the curve to be mapped on (re-displaying component as patches with different colors).

**Function Prototype:**

```c
void display_colorful_comp (comp_data *);
```

**Input Arguments:**

```c
comp_data *comp
```

pointer to data structure of component to be mapped on

**Output Arguments:**

None

**Function Output:**

None
B.6 `draw_curve.c`

B.6.1 `display_curve`

**Description:**

Gets all curve parts, opens structures, and sends for display.

**Function Prototype:**

```c
void display_curve (CURVE_MODEL *);
```

**Input Arguments:**

- `CURVE_MODEL *Curve_Model` pointer to CURVE_MODEL data structure

**Output Arguments:**

None

**Function Output:**

None

B.6.2 `draw_curve`

**Description:**

Generates points on curve and draws with polylines.

**Function Prototype:**

```c
void draw_curve (CURVE_DATA *);
```
Input Arguments:

CURVE_DATA *Curve  
pointer to CURVE_DATA data structure

Output Arguments:

None

Function Output:

None

B.6.3 find_knot

Description:

Finds the knot of the corresponding data point.

Function Prototype:

int find_knot (int, int, float [], int *);

Input Arguments:

int point  
point of interest
int num_knots  
number of knots defining the B-spline curve
float knot_sequence[]  
array of knot sequence

Output Arguments:

int *pre_knot  
previous knot of the corresponding knot of data point

Function Output:

int i  
corresponding knot
B.7 `draw_mapped_curve.c`

B.7.1 `draw_mapped_curve`

Description:

To draw the mapped curve on the surface.

Function Prototype:

```c
void draw_mapped_curve (MODEL *, CURVE_MODEL *, MAPPED_CURVE *[], comp_data *);
```

Input Arguments:

- `MODEL *Model`: pointer to MODEL data structure
- `CURVE_MODEL *Curve_Model`: pointer to CURVE_MODEL data structure
- `MAPPED_CURVE *Mapped_Curve[]`: pointer to MAPPED_CURVE data structure
- `comp_data *comp`: pointer to comp_data data structure

Output Arguments:

- `MAPPED_CURVE *Mapped_Curve[]`: pointer to MAPPED_CURVE data structure

Function Output:

None
B.7.2 display_all_mapped_curves

Description:

To display all curves mapped onto the model.

Function Prototype:

void display_all_mapped_curves (MODEL *, CURVE_MODEL *, MAPPED_CURVE *[]);

Input Arguments:

MODEL *Model
CURVE_MODEL *Curve_Model
MAPPED_CURVE *Mapped_Curve[]

pointer to MODEL data structure
pointer to CURVE_MODEL data structure
pointer to MAPPED_CURVE data structure

Output Arguments:

None
B.8 draw_patch.c

B.8.1 draw_patch

Description:
Main module to display a patch.

Function Prototype:
void draw_patch (comp_data *, PATCH_DATA *);

Input Arguments:
comp_data *comp pointer to comp_data data structure
PATCH_DATA *patch pointer to PATCH_DATA data structure

Output Arguments:
None

Function Output:
None

B.8.2 draw_u_line

Description:
Draws the \( u \) iso-parametric lines.

Function Prototype:
void draw_u_line (PATCH_DATA *, comp_data *, int, float);
Input Arguments:

PATCH_DATA *patch
comp_data *comp
int u_rng
float \( u \)

pointer to PATCH_DATA data structure
pointer to comp_data data structure
range of parametric \( u \) value
parametric value of \( u \)

Output Arguments:

None

Function Output:

None

B.8.3 draw_w_line

Description:

Draws the \( w \) iso-parametric lines.

Function Prototype:

void draw_w_line (PATCH_DATA *, comp_data *, int, float);

Input Arguments:

PATCH_DATA *patch
comp_data *comp
int w_rng
float \( w \)

pointer to PATCH_DATA data structure
pointer to comp_data data structure
range of parametric \( w \) value
parametric value of \( w \)

Output Arguments:

None

Function Output:

None
B.9 draw_temp_struct.c

B.9.1 display_temp_struct

Description:

To get structures for displaying $u$ and $v$ tangent lines, normal line, and the transformed curve.

Function Prototype:

```c
void display_temp_struct (PATCH_DATA *, CURVE_MODEL *);
```

Input Arguments:

- PATCH_DATA *picked_patch
- CURVE_MODEL *Curve_Model
  
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picked_patch</td>
<td>pointer to PATCH_DATA data structure</td>
</tr>
<tr>
<td>Curve_Model</td>
<td>pointer to CURVE_MODEL data structure</td>
</tr>
</tbody>
</table>

Output Arguments:

None

Function Output:

None

B.9.2 draw_transformed_curve

Description:

Generates points on the transformed curve and draws with polylines.

Function Prototype:

```c
void draw_transformed_curve (CURVE_DATA *);
```
Input Arguments:

CURVE_DATA *Curve    pointer to CURVE_DATA data structure

Output Arguments:

None

Function Output:

None

B.9.3 draw_projected_lines

Description:

To draw the transformed curve and its projected lines onto the surface.

Function Prototype:

void draw_projected_lines (MODEL *, CURVE_MODEL *);

Input Arguments:

MODEL *Model  pointer to MODEL data structure
CURVE_MODEL *Curve_Model  pointer to CURVE_MODEL data structure

Output Arguments:

None

Function Output:

None
B.10  map_curve.c

B.10.1  get_map_data

Description:

Gets input of necessary data from B-spline data structure to pass into intersection routine.

Function Prototype:

    void get_map_data (MODEL *, CURVE_MODEL *, comp_data *);

Input Arguments:

    MODEL *Model  pointer to MODEL data structure
    CURVE_MODEL *Curve_Model  pointer to CURVE_MODEL data structure
    comp_data *picked_comp  pointer to data structure of mapped component

Output Arguments:

    None

Function Output:

    None

B.10.2  get_bicubic_data

Description:

To get input suitable for sending into bicubic function.
Function Prototype:

    void get_bicubic_data (float [], float [], float [], [], float []);

Input Arguments:

    float u_knots
    float w_knots[]          u knots of patch
    float ctrlpt[4][4][3]    w knots of patch
    control points of patch

Output Arguments:

    float bicubics[3][4][4]    bicubic coefficients of patch

Function Output:

    None

B.10.3 map_intersect

Description:

    Gets points of intersection on the bicubic patches.

Function Prototype:

    void map_intersect (MODEL *, CURVE_MODEL *, comp_data *, MAPPED_CURVE *[]);

Input Arguments:

    MODEL *Model          pointer to MODEL data structure
    CURVE_MODEL *Curve_Model
    comp_data *comp        pointer to CURVE_MODEL data structure
    pointer to comp_data data structure

Output Arguments:

    None

Function Output:

    None
B.10.4 \hspace{1em} \textit{get\_edge\_point}

\textbf{Description:}

To check if mapped curve is out-of-bounds and hence if edge point is needed.

\textbf{Function Prototype:}

\begin{verbatim}
void get_edge_point (comp_data *, CURVE_DATA *, POINT_DATA **, POINT_DATA *,
                    POINT_DATA *, int, int [], int *);
\end{verbatim}

\textbf{Input Arguments:}

- \texttt{comp\_data *comp}
- \texttt{CURVE\_DATA *Curve}
- \texttt{POINT\_DATA **Point}
- \texttt{POINT\_DATA *newPoint}
- \texttt{POINT\_DATA *oldPoint}
- \texttt{int j}
- \texttt{int num\_line\_int[]}
- \texttt{int *int\_point}

\begin{itemize}
  \item pointer to \texttt{comp\_data} data structure
  \item pointer to data structure of curve part for processing
  \item double pointer to \texttt{POINT\_DATA} data structure
  \item pointer to data structure of the current point
  \item pointer to data structure of the previous point
  \item point being processed
  \item number of intersections of particular point (line) j
  \item counter of points that intersect
\end{itemize}

\textbf{Output Arguments:}

- \texttt{POINT\_DATA **Point}
- \texttt{int *int\_point}

\begin{itemize}
  \item double pointer to \texttt{POINT\_DATA} data structure
  \item counter of points that intersect
\end{itemize}

\textbf{Function Output:}

None

B.10.5 \hspace{1em} \textit{check\_corner\_intext}

\textbf{Description:}

To check if the four corners of the surface is internal or external to the curve.

\textbf{Function Prototype:}

\begin{verbatim}
void check_corner_intext (comp_data *, CURVE_DATA *, CORNER *[]);
\end{verbatim}

\textbf{Description:}

To check if the four corners of the surface is internal or external to the curve.
Input Arguments:

comp_data *comp
CURVE_DATA *Curve

pointer to component data structure
pointer to CURVE_DATA data structure

Output Arguments:

CORNER *Corner[]

pointer to CORNER data structure
1 - internal
2 - external

Function Output:

None

B.10.6 join_edge

Description:

To join the out-of-bounds curve with the appropriate edges of the surface to form a closed curve.

Function Prototype:

void join_edge (comp_data *, CURVE_DATA *, POINT_DATA **, POINT_DATA *,
POINT_DATA *, CORNER [], int *);

Input Arguments:

comp_data *comp
CURVE_DATA *Curve
POINT_DATA **Point
POINT_DATA *newPoint
POINT_DATA *oldPoint
CORNER *Corner[]
int *int_point

pointer to comp_data data structure
pointer to CURVE_DATA data structure
double pointer to POINT_DATA data structure
pointer to data structure of the current point
pointer to data structure of the previous point
pointer to surface corner data structure
number of intersections of a particular point (line)

Output Arguments:

POINT_DATA **Point

double pointer to POINT_DATA data structure

Function Output:

None
B.11  \textit{Model\_curve.c}

B.11.1  \textbf{Create\_NUBS3\_Curve}

\textbf{Description:}

Driver routine to create a non-uniform B-spline curve with specified constraints.

\textbf{Function Prototype:}

```c
void Create\_NUBS3\_Curve (int, float [], int, int [], float [], float **, float **, int *);
```

\textbf{Input Arguments:}

- \texttt{int numpts} \hspace{1cm} number of data points defining the curve
- \texttt{float datpts[]} \hspace{1cm} coordinates of data points
- \texttt{int param} \hspace{1cm} types of parametrization 1-uniform, 2-chord length, 3-centripetal
- \texttt{int continuities[]} \hspace{1cm} array of continuities at each point
- \texttt{float constraints[]} \hspace{1cm} array of constraints at each point in the form of tangent vectors

\textbf{Output Arguments:}

- \texttt{float **newknots} \hspace{1cm} knot vector array
  \hspace{1cm} (size = \texttt{numpts} + 6 \div 2 \times (\text{number of interior } C^0 \text{ points}) + (\text{number of interior } C^1 \text{ points}))

- \texttt{float **ctrlpts} \hspace{1cm} control points (size = \texttt{newknots} - 4)

\textbf{Function Output:}

None

\textbf{Author:}

S. Fleming
B.12  `process_picked_patch.c`

B.12.1  `process_picked_patch`

Description:

Processes the picked patch for mapping.

Function Prototype:

```c
void process_picked_patch (PATCH_DATA *, CURVE_MODEL *, comp_data *);
```

Input Arguments:

| PATCH_DATA *picked_patch | pointer to PATCH_DATA data structure |
| CURVE_MODEL *Curve_Model | pointer to CURVE_MODEL data structure |
| comp_data *picked_comp   | pointer to comp_data data structure |

Output Arguments:

None

Function Output:

None

B.12.2  `create_patch_marker`

Description:

To create a polymarker on the picked patch as a pointer.

Appendix B  C Source Code Functional Description  141
Function Prototype:

void create_patch_marker (PATCH_DATA *);

Input Arguments:

PATCH_DATA *picked_patch pointer to PATCH_DATA data structure

Output Arguments:

None

Function Output:

None

B.12.3 display_picked_patch

Description:

To display the picked patch.

Function Prototype:

void display_picked_patch (PATCH_DATA *);

Input Arguments:

PATCH_DATA *picked_patch pointer to PATCH_DATA data structure

Output Arguments:

None

Function Output:

None
B.12.4 new_marker_pt

Description:

To get the corresponding new marker point on patch with selection of pointer valuator button.

Function Prototype:

    void new_marker_pt (PATCH_DATA *, int, float *, float *, int, int);

Input Arguments:

    PATCH_DATA *picked_patch  pointer to PATCH_DATA data structure
    int ptr_vai_button          pointer valuator button selected
    float *u_loc                old u location of marker
    float *w_loc                old w location of marker
    int ustatus                 status of u pointer (increase, decrease or neither)
    int wstatus                 status of w pointer (increase, decrease or neither)

Output Arguments:

    float *u_loc                new u location of marker
    float *w_loc                new w location of marker

Function Output:

    None
B.12.5  move_marker_pt

Description:

To display the picked patch with the new marker point.

Function Prototype:

void move_marker_pt (PATCH_DATA *);

Input Arguments:

PATCH_DATA *picked_patch  pointer to PATCH_DATA data structure

Output Arguments:

None

Function Output:

None
B.13  \textit{ptr_val.c}

B.13.1  \texttt{draw_ptr_val}

Description:

To draw the software valuator buttons to move the patch pointer to serve as mapped point.

Function Prototype:

\begin{verbatim}
void draw_ptr_val (int *);
\end{verbatim}

Input Arguments:

None

Output Arguments:

\begin{verbatim}
int *root_id  structure id of pointer valuator buttons
\end{verbatim}

Function Output:

None
B.13.2     get_ptr_val

Description:

To get the pointer valuator button selected and move the marker accordingly.

Function Prototype:

void get_ptr_val (int, PATCH_DATA *, CURVE_MODEL *, comp_DATA *);

Input Arguments:

- int ptr_val_id: structure id of pointer valuator
- PATCH_DATA *picked_patch: pointer to data structure of patch picked
- CURVE_MODEL *Curve_Model: pointer to CURVE_MODEL data structure
- comp_data *picked_comp: pointer to comp_data data structure

Output Arguments:

None

Function Output:

None
B.14 read_curve.c

B.14.1 read_curve

Description:

To read in a curve data file and store in CURVE_MODEL data structure.

Function Prototype:

CURVE_MODEL *read_curve (CURVE_MODEL *, int *);

Input Arguments:

CURVE_MODEL *Curve_Model pointer to CURVE_MODEL data structure

Output Arguments:

int *Return 1 - successful read, 0 - unsuccessful read

Function Output:

CURVE_MODEL *Curve_Model pointer to CURVE_MODEL data structure

B.14.2 clean_curve

Description:

To free the memory space of old curve structure.

Function Prototype:

void clean_curve (CURVE_MODEL *);
Input Arguments:

CURVE_MODEL *Curve_Model  
pointer to CURVE_MODEL data structure

Output Arguments:

CURVE_MODEL *Curve_Model  
pointer to CURVE_MODEL data structure (reset, cleaned)

Function Output:

None

B.14.3  check_text

Description:

To check for the correct input text.

Function Prototype:

int check_text (FILE *, char *);

Input Arguments:

FILE *inp_file  
file pointer to the input file
char *text  
text to be checked

Output Arguments:

None

Function Output:

int strcmp()  
0 - correct, 1 - wrong
B.15 \textit{retrieve\_patch\_data.c}

B.15.1 \textit{retrieve\_patch\_data}

\textbf{Description:}

Retrieves patch data from component data.

\textbf{Function Prototype:}

\begin{verbatim}
void retrieve_patch_data (MODEL *);
\end{verbatim}

\textbf{Input Arguments:}

\begin{verbatim}
MODEL *Model pointer to MODEL data structure
\end{verbatim}

\textbf{Output Arguments:}

None

\textbf{Function Output:}

None

B.15.2 \textit{retrieve\_patch\_knots}

\textbf{Description:}

Retrieves the knots for a specific patch of a parametric surface direction.

\textbf{Function Prototype:}

\begin{verbatim}
void retrieve_patch_knots (float [], int, float []);
\end{verbatim}
Input Arguments:
float comp_knots[]
int patch_num

knot vector of the parametric surface direction of interest
patch number of the surface in this parametric direction

Output Arguments:
float patch_knots[]

the 8 knots of the patch in this parametric direction

Function Output:
None

B.15.3 corner_middle_data

Description:

Retrieves parametric (uw) and spatial (xyz) data at the lower patch corner (knot[3]), and the middle of
patch for define_point routine, and also at corners of uknot[4] and wknot[4].

Function Prototype:

void corner_middle_data (PATCH_DATA *);

Input Arguments:
PATCH_DATA *patch

pointer to PATCH_DATA data structure

Output Arguments:
None (calculated values stored in data structure)

Function Output:
None
B.15.4 cal_patch_point

Description:

Calculates xyz point at corner and middle of patch.

Function Prototype:

```c
void cal_patch_point (PATCH_DATA *, float [], float [], int, int, double []);
```

Input Arguments:

- PATCH_DATA *patch: pointer to PATCH_DATA data structure
- float Bu[]: blending functions of u
- float Bw[]: blending functions of w
- int u_rng: u range
- int w_rng: w range

Output Arguments:

- double point[3]: xyz point on patch

Function Output:

None
B.15.5 get_tangents

Description:


Function Prototype:

```c
void get_tangents (comp_data *, PATCH_DATA *);
```

Input Arguments:

```c
comp_data *comp
PATCH_DATA *patch
```

pointer to comp_data data structure

pointer to PATCH_DATA data structure

Output Arguments:

None (calculated values stored in data structure)

Function Output:

None
B.16  \textit{transform_curve.c}

B.16.1  \texttt{find_normal}

\textbf{Description:}

To get the normal vector of a B-spline patch on a specified point.

\textbf{Function Prototype:}

\begin{verbatim}
void find_normal (PATCH_DATA *, comp_data *, CURVE_MODEL *);
\end{verbatim}

\textbf{Input Arguments:}

\begin{tabular}{ll}
  PATCH_DATA *patch & pointer to PATCH_DATA data structure \\
  comp_data *comp & pointer to comp_data data structure \\
  CURVE_MODEL *Curve_Model & pointer to CURVE_MODEL data structure \\
\end{tabular}

\textbf{Output Arguments:}

None

\textbf{Function Output:}

None

B.16.2  \texttt{orient_curve}

\textbf{Description:}

To orient the curve model such that the plane defining it is normal to the normal vector found. The curve model will be normally projected onto the surface.
Function Prototype:

    void orient_curve (PATCH_DATA *, CURVE_MODEL *, double [], double [], double []);

Input Arguments:

    PATCH_DATA *patch  
    CURVE_MODEL *Curve_Model  
    double vect1[] 
    double vect2[]  
    double vect3[]  

pointer to PATCH_DATA data structure  
pointer to CURVE_MODEL data structure  
3 vectors on surface defining new system

Output Arguments:

    CURVE_MODEL *Curve_Model  

transformed control points of curve in CURVE_DATA data structure (passed back from transformed_curve routine)

Function Output:

    None

B.16.3  transform_curve

Description:

    To transform the curve part to the normal vector.

Function Prototype:

    void transform_curve (PATCH_DATA *, CURVE_DATA *, double [], double [], double []);

Input Arguments:

    PATCH_DATA *patch  
    CURVE_DATA *Curve  
    double normalized_x[]  
    double normalized_y[]  
    double normalized_z[]  

pointer to PATCH_DATA data structure  
pointer to CURVE_DATA data structure  
3 normalized orthogonal vectors on surface
Output Arguments:

CURVE_DATA *Curve  transformed control points of curve

Function Output:

None

B.16.4  flip_curve

Description:

To flip the curve to the opposite side of the surface by reversing normal vector's direction.

Function Prototype:

void flip_curve (PATCH_DATA *, CURVE_MODEL *);

Input Arguments:

PATCH_DATA *patch  pointer to PATCH_DATA data structure
CURVE_MODEL *Curve_Model  pointer to CURVE_MODEL data structure

Output Arguments:

None

Function Output:

None
B.16.5 rotate_curve

Description:
To rotate the curve 90 degrees.

Function Prototype:

    void rotate_curve (PATCH_DATA *, CURVE_MODEL *, int);

Input Arguments:

    PATCH_DATA *patch
    CURVE_MODEL *Curve_Model
    int count

    pointer to PATCH_DATA data structure
    pointer to CURVE_MODEL data structure
    counter of the number of rotation

Output Arguments:
None

Function Output:
None
B.17  util.c

B.17.1  normalize_vector

Description:

To normalize (conversion of a vector to a unit vector) a vector by replacing its components with direction cosines.

Function Prototype:

void normalize_vector (vector, vector);

Input Arguments:

vector a  the vector to be normalized\(^2\)

Output Arguments:

vector normalized  the normalized vector

Function Output:

None

B.17.2  matrix_mult

Description:

To perform a matrix multiplication.

---

\(^2\)vector is a user-defined type. It is an array ([3]) of type double.
Function Prototype:

void matrix_mult(matrix, matrix, matrix, int, int, int);

Input Arguments:

matrix matA  first matrix of size (row × row_col)\(^3\)
matrix matB  second matrix of size (row_col × col)
int row  number of rows of the first matrix
int row_col  number of columns of the first matrix which is also the number of rows of the second matrix
int col  number of columns of the second matrix

Output Arguments:

matrix matC  multiplied matrix of size (row × col)

Function Output:

None

B.17.3 find_nearest_point

Description:

To find the shortest distance from a series of points to a reference point.

Function Prototype:

int find_nearest_point(float [], float [], int);

Input Arguments:

float refpt[3]  the reference point
float point[][3]  points to be performed on [point number][xyz]
int num_pts  number of points

Output Arguments:

None

\(^3\)‘matrix’ is a user-defined type. It is a double pointer of type double.
Function Output:

    int nearer
    the nearest point number

---

**B.17.4 interpolation**

**Description:**

Interpolation function.

**Function Prototype:**

    float interpolation (float [], float [], float);

**Input Arguments:**

    float pt1[]
    pt1[0] - x value of first point, pt1[1] - y value of first point
    float pt2[]
    pt2[0] - x value of second point, pt2[1] - y value of second point
    float intpt_x
    x value of point to be interpolated

**Output Arguments:**

    None

**Function Output:**

    float intpt_y
    y value of point to be interpolated

---

**B.17.5 unnormalize**

**Description:**

To transform a normalized parameter back to the original parametrization.
Function Prototype:

float unnormalize (float, float *);

Input Arguments:

float knot parameter to be transformed
float *knot_seq knot sequence in the parametric direction

Output Arguments:

None

Function Output:

float transformed_knot the transformed knot

B.17.6 find_nearest_edge_knot

Description:

To find the nearest edge of surface as the edge point of curve if the curve is going out-of-bounds on the surface.

Function Prototype:

void find_nearest_edge_knot (comp_data *, int, float, float, float *, int *);

Input Arguments:

comp_data *comp pointer to data structure of component to be mapped on
int patch patch number of patch of intersection
float u u parameter of intersection point
float w w parameter of intersection point

Output Arguments:

float *edge nearest edge knot
int *const_param 1 - constant u, 2 - constant w

Function Output:

None
B.17.7 find_near_param

Description:

To find the closest edge knot if the curve is going out-of-bounds on one of the four corner patches of the surface.

Function Prototype:

    void find_nearest_param (float, float, float, float *, int *);

Input Arguments:

    float u                u parameter of intersection point
    float w                w parameter of intersection point
    float u_edge           edge knot of u (either uknot[3] or uknot[nu_knots-4])
    float w_edge           edge knot of w (either wknot[3] or wknot[nw_knots-4])

Output Arguments:

    float edge             nearest edge knot
    int const_param        1 - constant u, 2 - constant w

Function Output:

    None

B.17.8 minimum

Description:

    Gets the minimum of two values.

Function Prototype:

    float minimum (float, float);
Input Arguments:

\[
\begin{align*}
\text{float } x, y \quad &\text{the two values for comparison} \\
\end{align*}
\]

Output Arguments:

None

Function Output:

\[
\begin{align*}
\text{float } x \text{ or } y \quad &\text{minimum value of the two} \\
\end{align*}
\]

B.17.9  maximum

Description:

Gets the maximum of two values.

Function Prototype:

\[
\text{float maximum (float, float);} \\
\]

Input Arguments:

\[
\begin{align*}
\text{float } x, y \quad &\text{the two values for comparison} \\
\end{align*}
\]

Output Arguments:

None

Function Output:

\[
\begin{align*}
\text{float } x \text{ or } y \quad &\text{maximum value of the two} \\
\end{align*}
\]
B.17.10  traverse_point_data

Description:

Given the point number, all the points in the linked list are traversed to get the desired point entry.

Function Prototype:

POINT_DATA *traverse_point_data (CURVE_DATA *, int);

Input Arguments:

CURVE_DATA *Curve  pointer to CURVE_DATA data structure
int point_num      point number of desired point

Output Arguments:

None

Function Output:

POINT_DATA *newPoint  pointer to data structure of desired point in list
Vita

Jie Wen Tjung was born on February 16, 1967 in Medan, Indonesia, the country which he still maintains his permanent residency. He went to Penang, Malaysia, which is his country of citizenship, at the age of nine and lived with his twelve-year-old sister while attending elementary school. At the age of fourteen, he went to Singapore alone for his secondary education and lived in the Anglican House Hostel. He continued to move on to the United States for his college education after that. He attended Iowa State University for his freshman year, and continued on to finish up his Bachelor's degree in Mechanical Engineering at Oklahoma State University. Then he moved to Virginia Tech for his MSME degree. Upon graduation, he is prepared to embark on yet another journey. The author wishes to be together with his family to spend the Chinese New Year, which he had spent the occasion abroad for the last fifteen years.