2552
114

Data Reduction and Knot Removal for Non-Uniform B-Spline Surfaces

by

Fred W. Marcaly

thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

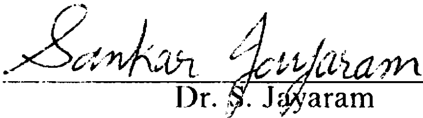
Master of Science

in

Mechanical Engineering

APPROVED:

Dr. Arvid Myklebust, Chairman



Dr. S. Jayaram

Dr. M. P. Deisenroth

April 18, 1991

Blacksburg, Virginia

c.2

LD
5655
V855
1991
M374
c.2

Abstract

B-Spline curves and surfaces are being used throughout the aircraft industry for geometric modeling. Geometric models having accurate surface representations in the non-uniform B-Spline surface format can contain very large quantities of data. The computing power required by a CAD system for visualization and analysis is directly influenced by these large amounts of data. Accordingly, a method for reducing the amount of data in a geometric model while maintaining accuracy is needed to reduce the computing power necessary to visualize and analyze a design. This thesis describes the refinement and implementation of a data reduction algorithm for non-uniform cubic B-Spline curves and non-uniform bi-cubic B-Spline surfaces. The topic of determining the significance of knots in non-uniform cubic B-Spline curves and non-uniform bi-cubic B-Spline surfaces is addressed. Also, a method for determining the order in which knots should be removed from non-uniform cubic B-Spline curves or non-uniform bi-cubic B-Spline surfaces during data reduction is presented. Finally, an algorithm for performing data reduction by removing knots from non-uniform cubic B-Spline curves and non-uniform bi-cubic B-Spline surfaces is presented.

Acknowledgements

I would like to thank my advisor, Dr. Arvid Myklebust, for giving me the opportunity to earn my Master of Science in Mechanical Engineering at Virginia Tech. I would also like to thank Dr. S. Jayaram and Dr. M. P. Deisenroth for being on my committee.

To Kris Kolady, Uma Jayaram, Eric V. Schrock, Michele Greishaber, J. R. Gloudemans, Bob Jones, and the rest of the ACSYNT group, thank you for your help. I wish you all the best.

Finally, I would like to thank my Mother, Father, and Brother for their unending encouragement through good times and bad.

Table of Contents

1.0 Introduction 1

1.1 Background 3

1.2 Objectives 4

1.3 Thesis Organization 5

2.0 Literature Survey 6

2.1 Fairing Methods for B-Splines 6

2.2 Knot Insertion Techniques 10

2.3 Knot Removal and Data Reduction for B-Splines 13

3.0 Spline Curves and Surfaces 16

3.1 B-Splines 16

3.1.1 B-Spline Nomenclature 17

3.1.2 Uniform Cubic B-Splines 19

3.1.3 Non-uniform Cubic B-Splines 22

3.1.4 Properties of the B-Spline Representation 25

3.2 B-Spline Surfaces 25

3.2.1 Uniform Bi-Cubic B-Spline Surfaces 26
3.2.2 Non-uniform Bi-Cubic B-Spline Surfaces 28
3.3 Discrete B-Splines 29

4.0 Data Reduction for Cubic B-Spline Curves 31
4.1 Difficulties Associated with Knot Removal 31
4.2 Overview of the Knot Removal Process 32
4.3 Classification of Knots for Removal 33
4.4 Calculation of New Control Vertices 40
4.5 A Data Reduction Algorithm for B-Spline Curves 45

5.0 Data Reduction for Bi-cubic B-Spline Surfaces 53
5.1 Classification of Knots for Removal from a B-Spline Surface 54
5.2 Calculation of New Control Vertices 54
5.3 Data Reduction for B-Spline Surfaces 55

6.0 Integration of Data Reduction into the ACSYNT B-Spline Module 56

7.0 Results 63
7.1 Non-Uniform B-Spline Curve Reduction 63
7.2 Surface Reduction 66

8.0 Conclusions and Recommendations 76
8.1 Conclusions 76
8.2 Recommendations 77

9.0 References 81

Appendix A. Norms **84**

A.1 Vector Norms 84

A.2 Norms Used for Knot Removal 86

Appendix B. Source Code Listing **87**

B.1 Classification of Knots for Removal 87

B.2 Calculation of New Control Vertices 110

B.3 Data Reduction for Non-Uniform Cubic B-Spline Curves 130

B.4 Data Reduction for Non-Uniform Bi-Cubic B-Spline Surfaces 140

B.5 Model Manipulation 151

B.6 Utilities 157

Vita **182**

List of Illustrations

Figure 1. Relationship Between a B-Spline Curve and Control Polygon 18
Figure 2. Curve Segments, Junction Points, and Control Vertices 21
Figure 3. Relationship Between Knots, Control Vertices and Points 24
Figure 4. Control Polyhedron for a B-Spline Surface 27
Figure 5. Steps in Determining the Significance of a Knot 35
Figure 6. Definition of a Local Cubic B-Spline While Removing One Knot 42
Figure 7. Local Cubic B-Splines Considered When Removing a List of Knots . . . 46
Figure 8. Local Cubic B-Splines Considered When Removing a List of Knots . . . 47
Figure 9. Removal of Knots Not Affecting the Same Local Region of the Curve . 48
Figure 10. Modified Model Data Structure 58
Figure 11. Modified Component Data Structure 59
Figure 12. Removal Data Structure for Storing Weights 60
Figure 13. Weight Data Structure Used During Classification of Knots 61
Figure 14. Approximation Data Structure Used During Knot Removal 62
Figure 15. Original Airfoil Curve, 104 Knots and 100 Control Vertices 64
Figure 16. Reduced Airfoil Curve, 54 Knots and 50 Control Vertices 65
Figure 17. Original I-Beam Curve, 405 Knots and 401 Control Vertices 67
Figure 18. Reduced I-Beam Curve, 325 Knots and 321 Control Vertices 68
Figure 19. Reduced I-Beam Curve, 245 Knots and 241 Control Vertices 69

Figure 20. Original Parabolic Surface, 75 x 75 Vertex Control Polyhedron 70
Figure 21. Intermediate Parabolic Surface, 12 x 75 Vertex Control Polyhedron ... 72
Figure 22. Reduced Parabolic Surface, 12 x 12 Vertex Control Polyhedron 73
Figure 23. Original Wing Surface, 25 x 100 Vertex Control Polyhedron 74
Figure 24. Reduced Wing Surface, 16 x 50 Vertex Control Polyhedron 75

1.0 Introduction

Engineering design can be divided into three steps: conceptual design, preliminary design and final design. Throughout the process the designer may need to visualize the design to get a better understanding of how parts fit together or interact during service. In many instances the designer is interested in the look or feel of a design. This is especially true in the aircraft and automotive industries, where look and feel are important characteristics of a product.

Geometric modeling is one method which is commonly used to view designs at the conceptual level. Geometric modeling consists of generating, manipulating, storing and displaying a mathematical approximation of the shape of an object. Geometric models are often used to generate engineering drawings or computer graphics displays. The amount of data involved in a geometric model can be overwhelming. Accordingly, the amount of information initially used to represent surfaces in a geometric model can be rather large and result in degraded performance of computer-aided design (CAD) software.

Design at the preliminary level requires more specific data than at the conceptual level. Geometric models are often manipulated to find engineering data such as mass properties, surface areas, weights, or parametric information about a design. In addition to generating data of this type, geometric models are commonly used to transfer designs from one CAD application to another. In aircraft design, for instance, mesh generation for Computational Fluid Dynamics (CFD) codes and Radar Cross-Section (RCS) codes can be done using geometric surface models. However, both CFD and RCS codes require positional, gradient and curvature continuity (c^2 continuity) between surfaces representing the aircraft shape. B-Spline curves and surfaces satisfy these requirements and are being used throughout the aircraft industry. The use of B-Spline curves and surfaces for accurate surface representations can result in geometric models containing very large quantities of data. As mentioned above for CAD software, these extreme amounts of data can hamper the performance of codes used to find engineering properties of a design or generate meshes for CFD and RCS codes.

The computing power required by a CAD system is influenced by the amount of data in a geometric model. Therefore models having less data are preferable. Accordingly, a method for reducing the amount of data in a geometric model while maintaining accuracy is necessary to reduce the computing power needed to visualize and analyze a design.

1.1 Background

Several design codes exist for the conceptual design of aircraft. One such code is ACSYNT (AirCraft SYNThesis). ACSYNT was developed during the early 1970s at NASA Ames Research Center [Greg73]. The initial goal of ACSYNT was to develop an analysis tool that was flexible enough to analyze both military and civil aircraft. To help accomplish this goal, the code was parameter based to provide for easy entry of aircraft characteristics, including both geometry and performance. However, ACSYNT was based on batch processing and did not provide the designer with a good means of visualizing the design.

In 1987 the Virginia Polytechnic Institute and State University CAD laboratory began the development of a CAD system for ACSYNT. The goal of the CAD system, designed under the direction of Dr. Arvid Myklebust, was to enhance the design capabilities of ACSYNT through an interactive graphical interface and a contoured surface model. The graphical interface allowed the user to visualize both input and output. The interface used the 3D graphics standard Programmers Hierarchical Interactive Graphics System (PHIGS). Use of PHIGS provided for device independence [Wamp88a, Wamp88b]. ACSYNT represented aircraft geometry using components such as the nose, wing, horizontal tail, etc. The components were modeled using the bi-cubic Hermite surface representation. This method was very good for displaying and manipulating wireframe and shaded images of the aircraft geometry. The bi-cubic Hermite surface representation also offered both positional and gradient continuity, but did not provide curvature continuity along component surfaces. Accordingly, a B-Spline module was added to ACSYNT to allow curvature continuity along component surfaces.

The new module used a non-uniform bi-cubic B-Spline surface representation for aircraft components and allowed for the calculation of intersections between individual components. Also, filleting (or blending) of surfaces between components was added to allow the designer to construct entirely curvature continuous models. At the same time, neutral file exchanges using the Initial Graphics Exchange Specification (IGES) for geometry data were added.

1.2 Objectives

Although these improvements have brought ACSYNT closer to being easily interfaced with other design systems, additional research and development is still needed. In particular, to use geometric models in the B-Spline representation from other design systems, a method for utilizing large models is necessary. The size of models in the B-Spline format which are expected to be imported from other systems tends to be very large. Therefore, the computing time needed to display both wireframe and shaded images of the models impedes the design process. Also, if these models are to be exported from ACSYNT to other applications, such as CFD or RCS codes, the amount of data must be reduced. Accordingly the objective of this thesis is to implement data reduction and knot removal for non-uniform tensor product B-Spline surfaces in the B-Spline module of ACSYNT while maintaining the accuracy of the models.

1.3 Thesis Organization

The remainder of this thesis is divided into the following sections:

- Literature survey,
- B-Spline curves and surfaces,
- Data reduction for cubic B-Spline curves,
- Data reduction for bi-cubic B-Spline surfaces,
- Integration of data reduction into the ACSYNT B-Spline module,
- Results, and
- Conclusions and Recommendations.

Also a code listing for routines unique to the data reduction algorithm implemented in the ACSYNT B-Spline module is given in Appendix A. For additional information, refer to to ACSYNT B-Spline module manual.

2.0 Literature Survey

2.1 Fairing Methods for B-Splines

Fairing of spline curves refers to the process of changing the data defining a spline so that the curve will look more "pleasant", "sweet" or "fair". From a designer's point of view, a fair curve will require a relatively small number of French curves to draw. In the mathematical sense, the "fairness of a curve amounts to requiring that its curvature be almost piecewise linear, with only a small number of segments. Continuity of curvature is an obvious additional requirement" [Fari87]. Fairing of splines has been done on curves in both the Hermite representation and in the B-Spline representation. In general, fairing involves removing a knot (or data point) and reinserting it back into the curve to give a more desirable shape. Accordingly, for the purpose of investigating knot removal and data reduction, the first step of the fairing process, removing a knot, is of interest.

Most of the current research on fairing splines has been based on an algorithm used to fair cubic Hermite curves. The fairing algorithm involves improving the shape of the curve by changing the location of a junction point [Kjel83]. The data point defining the junction point between two curve segments is first removed from the curve, then reinserted to give a more desirable shape. This method of fairing is a global algorithm because it affects a large portion of the curve. Since the algorithm is based on the Hermite representation and affects the curve globally, the process for knot removal certainly can not be applied to data reduction for B-Splines.

Fairing of B-Spline curves was first addressed by Farin, Rein, Sapidis, and Worsey [Fari87]. An algorithm for locally fairing B-Spline curves to produce a pleasant curvature plot is presented in "Fairing Cubic B-Spline Curves". The problem which is addressed is that of modifying a given c^2 cubic B-Spline curve locally so that the resulting curve is fairer than the original at a given knot. The proposed solution involved two steps:

1. Remove the knot from the knot sequence. Then find a new control polygon which approximates the original control polygon.

2. Using the new control polygon, reinsert the knot so that the new curve is defined over the original knot sequence.

Three knot removal methods are suggested by Farin [Fari87]. All three of these methods are based on a process which is best described as being the inverse of knot insertion. The first two methods involve the use of extrapolation to determine new control points. However, both extrapolation methods required the selection of a free parameter. The selection of this free parameter was difficult to intrinsically code on a computer. The

third knot removal method involved finding a least squares solution for the new control vertices using matrix methods. The least squares method minimizes the change in distance between the control vertices remaining in the control polygon. Accordingly, this method gave the best results.

Although these methods remove a knot with varying degrees of accuracy of the resulting curve, for the purpose of data reduction, all three methods have major draw backs. The first difficulty which would be encountered with implementing these methods in a data reduction algorithm is that all three methods can remove only one knot at a time. This results in the need to perform the same algorithm repeatedly to remove a large number of knots from an unwieldy curve. The second difficulty encountered in including these methods in a data reduction algorithm is the inability to retain the original convexity of the curve from which a knot is being removed. The combination of being able to only remove one knot at a time and the uncertainty of whether the original convexity of the curve will be retained resulted in none of these methods being used in the data reduction algorithm implemented for this thesis.

Another important part of fairing a B-Spline curve is the selection of the knots at which to fair the curve. Likewise, selecting which knots should be removed from a curve during data reduction is an important part of the knot removal problem. Farin [Fari87] suggests three ways of selecting knots at which the curve should be faired:

- Find the largest discontinuity in the third derivative and fair the curve at that knot,
- Find the largest rate of change in the first derivative and fair the curve at this knot, or

- Leave the choice of where to fair the curve up to the trained eye of the designer.

Certainly, these criteria for fairing are not meant to be used for data reduction, but the first two items are of interest. Since the objective of this thesis is to remove knots from a surface while maintaining accuracy of the surface, the first two criteria may or may not be appropriate for choosing the knots that should be removed. In later sections, the use of these criteria for choosing which knots should be removed during data reduction will be more thoroughly discussed.

More recently, Farin [Fari90a] presents a knot removal technique in "Automatic Fairing Algorithm for B-Spline Curves" which maintains the original convexity of the curve better than the methods presented in his first paper on fairing B-Splines. This is the first automatic fairing algorithm to appear in the literature. Farin automated the fairing process by using the criterion that a curve should always be faired at the largest curvature fluctuation. This is referred to as the "curvature-correction" algorithm. The algorithm included a step to ensure that the convexity of the curve was retained by comparing the convexity of the new control polygon to the convexity of the original control polygon. Again, this algorithm is meant to remove only one knot from the curve at a time. While the algorithm can be applied to several neighboring knots or multiple knots, the entire algorithm must be repeated as each knot is removed. The curvature-correction algorithm can only be used on curves, however extending curvature-correction to surfaces is labeled as a problem of considerable interest.

2.2 *Knot Insertion Techniques*

Knot removal can to some extent be thought of as the inverse of knot insertion, except that choosing which knot to remove is usually more difficult than choosing where to insert a knot. Accordingly, past research done on knot insertion may give some insights into the knot removal process. De Boor [DeBo72] published the first algorithm on knot insertion for B-Splines. This algorithm has been refined by Boehm [Boeh80], [Boeh85]. While De Boor discussed knot insertion only for curves, the algorithm given by Boehm can be applied to both curves and surfaces. Boehm's method of knot insertion is very reliable, but only provides for the insertion of one knot at a time. To insert several knots, the algorithm must be repeated for each new knot.

Perhaps the most general knot insertion or refinement algorithm for B-Splines is the Oslo algorithm developed by Cohen, Lyche, and Riesenfeld [Cohe80]. The Oslo algorithm allows more than one knot to be inserted into a B-Spline curve in a single iteration. This method of knot insertion is based on the fact that a curve constructed from one set of control vertices, blending functions and knots can be represented in terms of a larger set of control vertices, new blending functions and a refined set of knots. In other words, the knot insertion process produces a B-Spline space based on a refined knot sequence which contains the original B-Spline space based on the original knot sequence [Boeh85]. Therefore, to determine the new B-Spline space from the original B-Spline space, a transformation based on linear algebra can be used.

Bartels [Bart87] gives a good explanation of the Oslo algorithm. Based on the above discussion of knot insertion, a curve constructed:

- from an original set of control vertices, $\mathbf{V}_0, \ldots, \mathbf{V}_m$,
- weighted by an original set of B-Spline blending functions, B_i, of order k,
- and defined on an original knot sequence, $\{\tau_j\}_0^{m+k}$

can be represented in terms of:

- a larger set of control vertices, $\mathbf{W}_0, \ldots, \mathbf{W}_{m+n}$,
- weighted by a refined set of B-Spline blending functions, N_j, also of order k,
- and defined on a refined knot sequence, $\{t_j\}_0^{m+n+k}$.

Accordingly, a transformation from the original control vertices, $\mathbf{V}_i$, to the refined control vertices, $\mathbf{W}_i$, can be expressed as:

$$\mathbf{W}_j = \sum_{i=0}^{m} \alpha_{i,k}(j) \mathbf{V}_i \quad for \;\; j = 0, \ldots m+n$$

where the $\alpha_{i,k}$ are discrete B-Splines of the same order and having the same knot sequence as the B-Spline into which knots are being inserted.

In a similar manner, a transformation from the original set of B-Splines, B_i, to the refined set of B-Splines, N_j, can be expressed as:

$$B_{i,k}(t) = \sum_{j=1}^{m+n} \alpha_{i,k}(j) N_{j,k}(\tau)$$

The B-Splines can be expressed as the vector spaces:

$$B(x) = (B_0(x), \ldots, B_m(x))^T$$

$$N(x) = (N_0(x), \ldots, N_{m+n}(x))^T$$

Using this notation the linear transformation from B to N can be expressed in matrix form as:

$$\mathbf{N}(x) = \mathbf{A}^T \mathbf{B}(x)$$

The matrix $\mathbf{A}$ is called the B-Spline knot insertion matrix of order k from τ to t. The knot insertion transformation can be expressed in expanded matrix form as:

$$f(x) = \sum_{i=0}^{m} \mathbf{V}_i B_{i,k}(\tau) = \mathbf{V}^T \mathbf{B}(x) = \mathbf{V}^T \mathbf{A}^T \mathbf{N} = \sum_{j=0}^{m+n} \mathbf{W}_j N_{j,k}(t)$$

Based on the linear independence of B-Splines, the transformation can be applied directly to the control vertices to yield:

$$\mathbf{W} = \mathbf{A}\mathbf{V}$$

The elements of the B-Spline knot insertion matrix are discrete B-Splines of the same order, k, as the B-Spline into which knots are being inserted.

The Oslo algorithm has two beneficial characteristics, one of which is essential if the inverse of the knot insertion process is to be used as the knot removal process. This method of knot insertion provides for inserting more than one knot in a single iteration. This feature of the Oslo algorithm should allow for a more efficient process if the algorithm is to be inverted for use in knot removal. Similar to Boehm's method of knot insertion, the Oslo algorithm only affects the curve locally.

2.3 Knot Removal and Data Reduction for B-Splines

The first method for removing knots from a B-Spline using the B-Spline knot insertion matrix is given by Lyche and Morken [Lych87a] in a paper entitled "A Discrete Approach to Knot Removal and Degree Reduction Algorithms for Splines". Lyche and Morken address the problem of computing a B-Spline approximation to a piecewise polynomial given as a linear combination of B-Splines while retaining the same order. Lyche considers two B-Splines, f and g, of the same order having different knot sequences where g is to approximate f. The difference between these B-Splines, $f-g$, is shown to be a B-Spline of the same order which has a knot sequence containing the knot sequences of both f and g. The difference in the B-Splines, $f-g$, can be minimized by varying the approximating B-Spline, g, over the linear space defined by its knot sequence. While this method allows more than one knot to be removed from a B-Spline curve in a single iteration, a robust algorithm for data reduction is not discussed.

A robust algorithm for data reduction is given in "A Data Reduction Strategy for Splines with Applications to the Approximation of Functions and Data" [Lych88]. In this paper, Lyche and Morken present a technique for reducing the number of knots in a B-Spline curve without disturbing the curve by more than a certain tolerance. This technique for removing knots is based on the work done earlier by Lyche et al.

Lyche considers a B-Spline, $g \in \mathbf{S}_{k,\tau}$, which approximates another B-Spline $f \in \mathbf{S}_{k,t}$ where $\mathbf{S}_{k,t}$ and $\mathbf{S}_{k,\tau}$ are vector spaces defined by B-Spline blending functions. Also, a tolerance, ε, for the difference between the two B-Splines is given. The problem which is addressed is to find an approximation, g, such that $||f - g|| < \varepsilon$. The algorithm used to solve this problem is divided into three steps referred to as ***rank***, ***remove*** and ***approximate***. In ***rank***, the order in which knots should be removed from the original knot sequence is determined. In ***remove***, the number of knots which can be removed from the original knot sequence without exceeding the tolerance, ε, is determined. Then a reduced knot sequence is created as a subsequence of the original knot sequence. In ***approximate***, the control vertices for the approximating B-Spline, g, are found using a least squares approximation.

The knot removal strategy for B-Spline curves outlined above is extended to tensor product surfaces in "Knot Removal for Parametric B-Spline Curves and Surfaces" [Lych87b]. Lyche and Morken propose that one way to perform data reduction on a tensor product B-Spline surface is to perform data reduction on the curves defining the surface in each parametric direction. Accordingly, the technique outlined in "A Data Reduction Strategy for Splines with Applications to the Approximation of Functions and Data" [Lych88] is used first in one parametric direction and then in the other parametric direction on each isoparametric curve defining the control polyhedron of the

surface. This algorithm has been proven to be robust for knot removal and data reduction on tensor product B-Spline surfaces.

3.0 Spline Curves and Surfaces

Spline curves are commonly used in the aircraft and shipbuilding industries [Mort85]. The spline curve originates from a drafting tool, called a spline, which is a strip of plastic, wood or other material that is flexed to pass through a series of design points. The most important characteristic of a spline curve is that the curvature is continuous. This results in curves having no kinks or abrupt changes in shape. Although there are several different representations for splines, including Hermite and Bezier, B-Splines have become popular for use in geometric modeling.

3.1 B-Splines

A B-Spline curve is an approximation of the points defining a control polygon. The curve may or may not pass through the points defining the vertices of the control polygon. A B-Spline curve and the corresponding control polygon are shown in

Figure 1 on page 18. A point on the curve is a weighted average of several points in the control polygon. The number of points influencing the weighted average is equal to the order of the curve. The weighted average is based on blending functions which control how much each control vertex affects the points on the curve. The blending functions are determined from the knot sequence for the curve. The remaining sections in this chapter review the relationship between the control polygon, knot sequence, blending functions, and points on the curve.

3.1.1 B-Spline Nomenclature

Information on B-Splines is available through out the literature [Fari90b], [Yama88], [Bart87]. However, there is often a difference in the nomenclature used in the description of B-Splines. The format adopted for this thesis is based on a series of papers written by Lyche and Morken [Lych87a], [Lych87b], [Lych88]. All equations for B-Spline curves and surfaces given in this thesis are for fourth order (third degree) B-Splines unless otherwise stated. The symbols used to represent B-Splines in equation form and their definitions are given below:

Symbol	***Definition***
k	Order of B-Spline (for third degree, $k = 4$)
p	Point on the B-Spline
q, c, d	Control vertex
Q	Control polygon or polyhedron
u, w, t	Knot values along the B-Spline
T	Knot sequence for the entire B-Spline
N, B	Blending or B-Spline basis functions

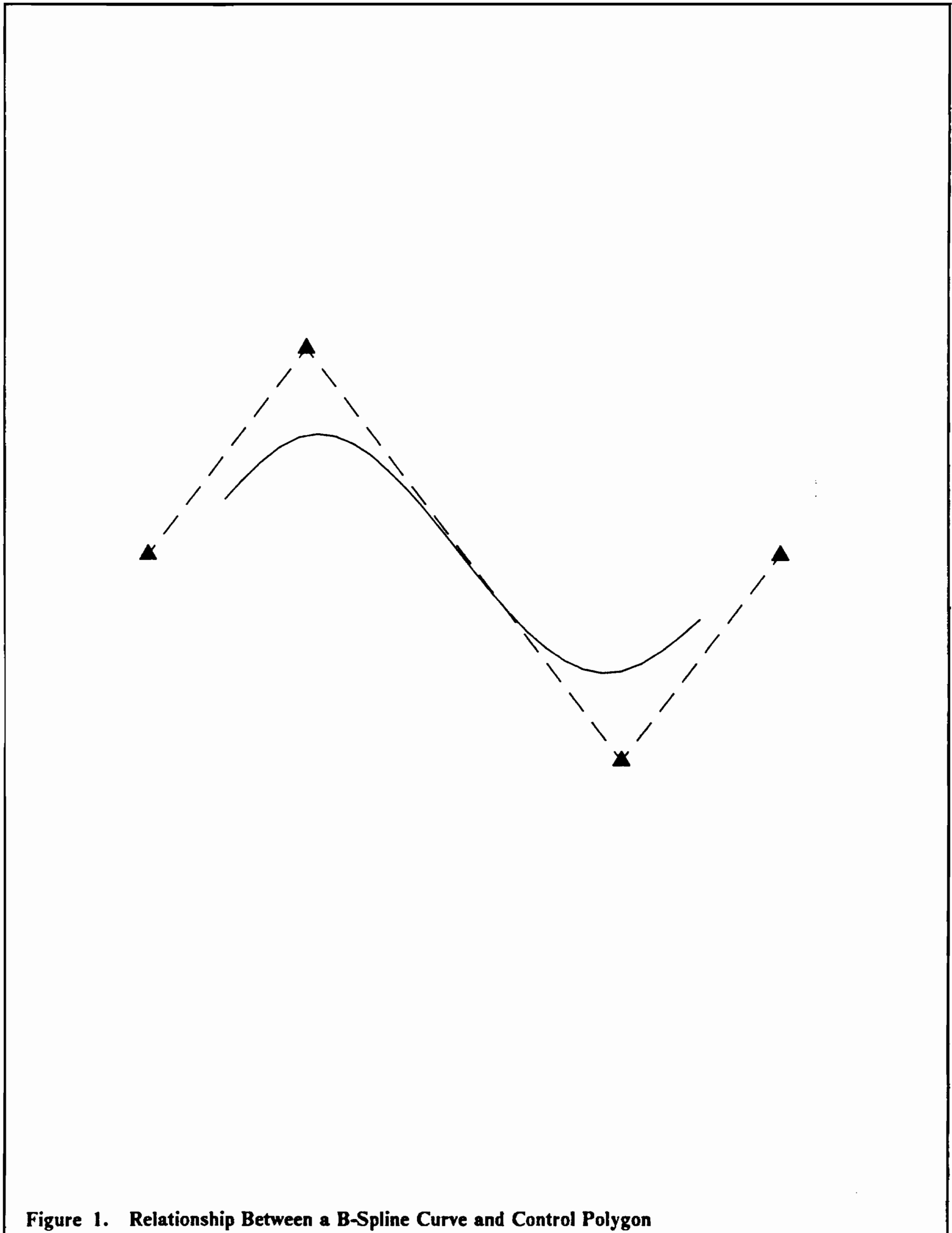


Figure 1. Relationship Between a B-Spline Curve and Control Polygon

3.1.2 Uniform Cubic B-Splines

The equation for a uniform cubic B-Spline is given below:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{q}_i N_{i,k}(u)$$

This equation shows that any point on a B-Spline curve is the weighted average of the surrounding control vertices. In the above equation, $\mathbf{q}_i$ refers to control vertices, while $N_{i,k}$ refers to blending functions. For any B-Spline, no more than k blending functions are non-zero for any point on the curve. This feature results in local control which is characteristic of B-Splines. The term uniform refers to the spacing between knot values. Accordingly, the difference between any two successive knot values, $t_{i+1} - t_i$, is constant for a uniform B-Spline. As a result, a uniform B-Spline curve has the same blending functions over the entire curve.

For a uniform cubic B-Spline ($k = 4$), any point on the curve is affected by no more than four control vertices. Thus, the smallest component of a cubic B-Spline curve, a curve segment, must have at least four control vertices. The equation of a uniform cubic B-Spline expressed in matrix form is:

$$P_i(u) = UMQ_i$$

where:

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad Q_i = \begin{bmatrix} \mathbf{q}_{i-1} \\ \mathbf{q}_i \\ \mathbf{q}_{i+1} \\ \mathbf{q}_{i+2} \end{bmatrix}$$

The so called universal transformation matrix, M, is the matrix of blending function coefficients over the entire curve. The product of the U matrix and the universal transformation matrix is used to find the blending functions at any point on a uniform cubic B-Spline curve.

Most B-Spline curves used in design applications are composed of several curve segments. For example, since a cubic B-Spline curve segment is defined by only four control vertices, a B-Spline curve having eight control vertices is composed of five curve segments. This is illustrated in Figure 2 on page 21. A curve segment usually starts at a local parameter value of zero and ends at a local parameter value of one. The blending functions vary based on the value of the parameter, u. As the local parameter changes from zero to one along the length of a curve segment, the amount that each control vertex influences the curve changes.

A junction point is defined as a point on a B-Spline curve where two curve segments join, while a break point is the knot corresponding to a junction point. Two junction points appear on the curve in Figure 2 on page 21 and are labeled p_1 and p_2. Junction points can also be defined as the point at which the local parameter value is one on the preceding curve segment and the point at which the local parameter value could be zero on the next curve segment. However, it is important to note that when the local parameter value is zero, the curve segment is only affected by the first three control

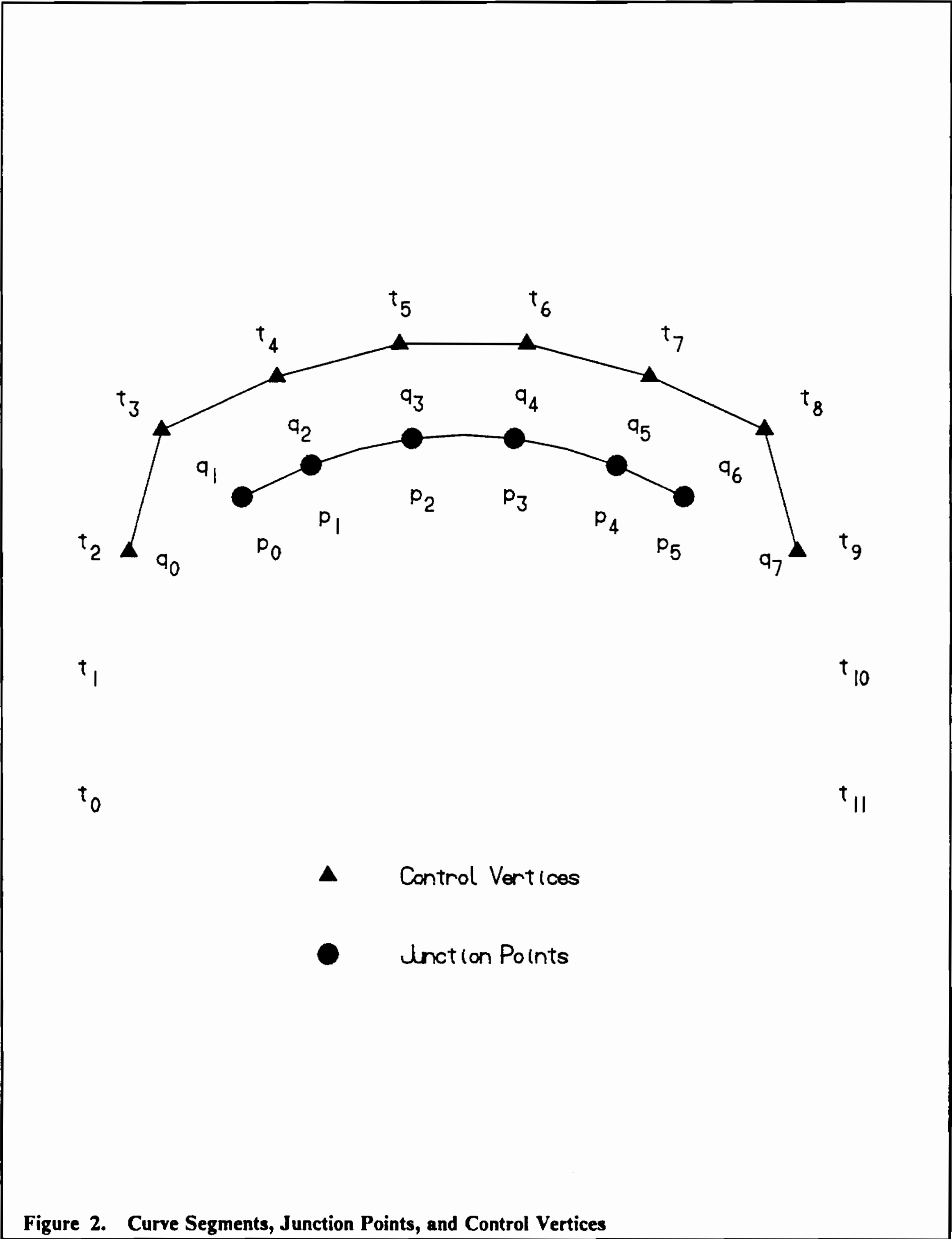


Figure 2. Curve Segments, Junction Points, and Control Vertices

vertices. Similarly, when the local parameter value is one, the curve segment is only affected by the last three control vertices. This, in effect, defines the transition from one set of control vertices affecting the curve to the next set of control vertices affecting the curve.

A typical knot sequence for a uniform B-Spline curve of order k with m control vertices is $\mathbf{t} = \{t_i\}_{i=1}^{m+k} = \{-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8\}$. This knot sequence could be used for the curve in Figure 2 on page 21. This is referred to as a uniform knot sequence because the change in knot values from one knot to the next is constant. Knot sequences for B-Spline curves are always non-decreasing. In the above knot sequence, interior knots are defined to be:

$$t_i \ni t_k \leq t_i \leq t_m$$

Accordingly, in Figure 2 on page 21, knots $t_4, \ldots, t_7$ are interior knots.

3.1.3 Non-uniform Cubic B-Splines

The equation for a non-uniform cubic B-Spline is the same as for a uniform cubic B-Spline. However, non-uniform cubic B-Splines offer better shape control than uniform cubic B-Splines. This improvement in shape control results from the knot sequence. As the name implies, the difference between any two successive knot values of a non-uniform cubic B-Spline is not necessarily the same. This difference is called the knot spacing. The knot spacing influences how much each control vertex affects the curve. Two knot sequences having the same knot spacing can be used with the same

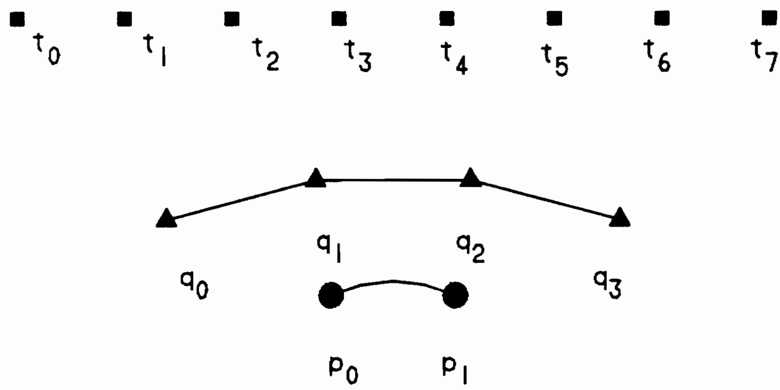
control polygon to define equivalent curves. Accordingly, a knot sequence of $\{1, 3, 7, 8\}$ has the same effect as a knot sequence of $\{5, 7, 11, 12\}$ when used with the same control polygon. The relationship between the knot sequence, control polygon and points along a non-uniform B-Spline curve is shown in Figure 3 on page 24. In this thesis, one knot value is used for each control vertex as well as two additional knot values at each end of the control polygon.

The blending functions for non-uniform B-Spline curves are calculated using a recursive formula which is the sum of divided differences of the knot sequence. The blending functions are found using:

$$N_{i,1}(u) = \begin{Bmatrix} 1 & if\ t_i \leq u < t_{i+1} \\ 0 & otherwise \end{Bmatrix}$$

$$N_{i,k} = \frac{(u - t_i)\, N_{i,k-1}(u)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - (u))\, N_{i+1,k-1}(u)}{t_{i+k} - t_{i+1}}$$

Similar to uniform B-Splines, only k or fewer blending functions will be non-zero, resulting in no more than k control vertices affecting any point along the B-Spline curve. The discussion on curve segments and junction points also applies to non-uniform B-Splines. The parameter values at the beginning and end of a curve segment are equal to the knot values of the interior knots of the control polygon for that curve segment. Even so, local parameter values can range from zero to one.



● Number of Junction Points = n (2)
▲ Number of Control Vertices = n+2 (4)
■ Number of Knots = n+6 (8)

Correspondence

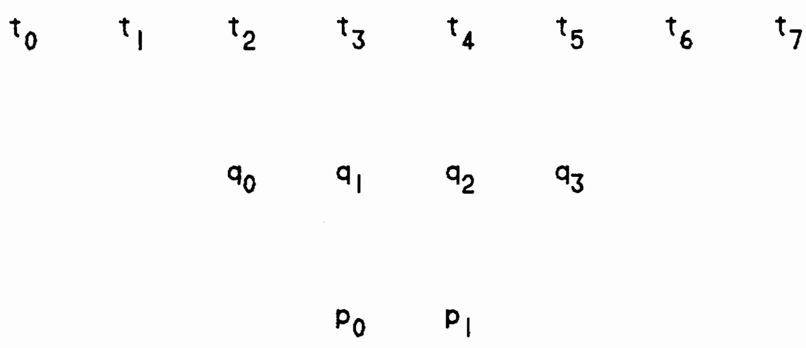


Figure 3. Relationship Between Knots, Control Vertices and Points

3.1.4 Properties of the B-Spline Representation

There are several properties of B-Splines which must be considered when implementing a knot removal or data reduction algorithm including:

- C^{k-1} Continuity Between Curve Segments: A cubic curve guarantees c^2 continuity at curve junction points.

- Variation Diminishing Property: A B-Spline curve never intersects any arbitrary straight plane more than its control polygon does. Therefore, the shape of the control polygon is reflected in the shape of the B-Spline curve.

- Local Shape Control: The effect of any control vertex is limited to k curve segments. Accordingly, any point along a cubic B-Spline curve is affected by at most four control vertices.

- Curve Degree Control: The degree of a B-Spline curve is independent of the number of control vertices.

3.2 B-Spline Surfaces

The representation of B-Spline surfaces is similar to that for B-Spline curves. A B-Spline surface is an approximation to a lattice of control vertices, commonly called a control polyhedron. While having two parametric directions instead of one, B-Spline surfaces

have the same properties as curves. The concepts of curve segments and junction points for curves can be extended to B-Spline surfaces, but because of the existence of two parametric directions, these become patches and junction curves, respectively. The general expression for a B-Spline surface is given below:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{q}_{i,j} \, N_{i,M}(u) \, N_{j,L}(w)$$

Two parametric directions, u and w, and two knot sequences are used instead of one. In the above equation, $\mathbf{q}_{i,j}$ represents control vertices, while $N_{i,M}$ and $N_{j,L}$ stand for the blending functions for each parametric direction. The terms M and L stand for the order of the B-Spline surface in the u and w parametric directions, respectively. For a bi-cubic B-Spline surface, $M = L = 4$. A control polyhedron for a B-Spline surface is shown in Figure 4 on page 27.

3.2.1 Uniform Bi-Cubic B-Spline Surfaces

Similar to uniform cubic B-Spline curves, the coefficients of the blending functions for uniform bi-cubic B-Spline surfaces are constants. The simplified form of a uniform bi-cubic B-Spline surface follows:

$$p_{i,j}(u,w) = U \, M \, Q \, M^T \, W^T$$

where:

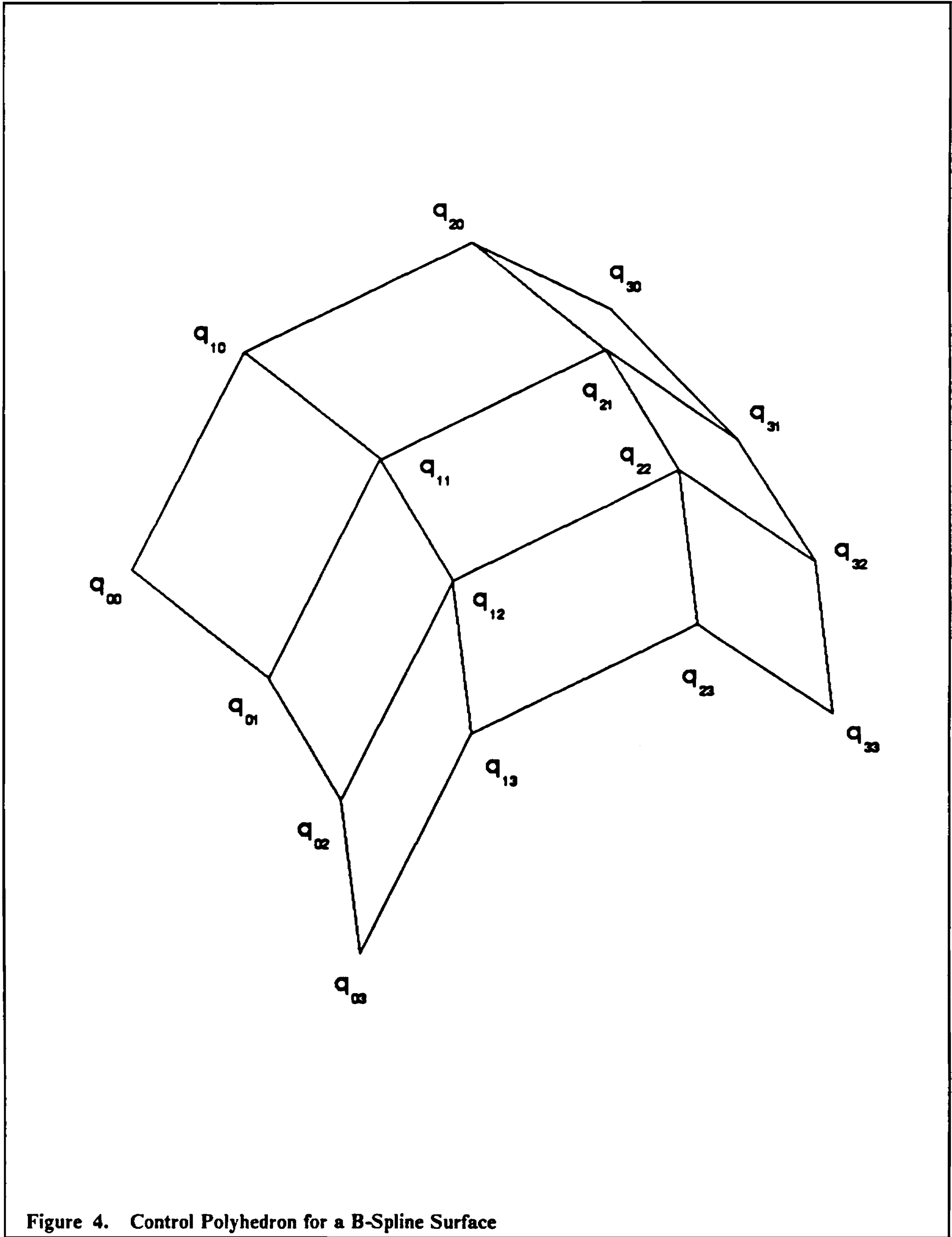


Figure 4. Control Polyhedron for a B-Spline Surface

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad W = \begin{bmatrix} w^3 & w^2 & w & 1 \end{bmatrix}$$

$$M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \qquad Q = \begin{bmatrix} \mathbf{q}_{i-1,j-1} & \mathbf{q}_{i-1,j} & \mathbf{q}_{i-1,j+1} & \mathbf{q}_{i-1,j+2} \\ \mathbf{q}_{i,j-1} & \mathbf{q}_{i,j} & \mathbf{q}_{i,j+1} & \mathbf{q}_{i,j+2} \\ \mathbf{q}_{i+1,j-1} & \mathbf{q}_{i+1,j} & \mathbf{q}_{i+1,j+1} & \mathbf{q}_{i+1,j+2} \\ \mathbf{q}_{i+2,j-1} & \mathbf{q}_{i+2,j} & \mathbf{q}_{i+2,j+1} & \mathbf{q}_{i+2,j+2} \end{bmatrix}$$

3.2.2 Non-uniform Bi-Cubic B-Spline Surfaces

While the general format for a B-Spline surface can certainly be applied to a non-uniform bi-cubic B-Spline surface, the matrix form, shown below, is sometimes used:

$$\mathbf{p}_{i,j}(u,w) = \begin{bmatrix} N_{0,4}(u), N_{1,4}, N_{2,4}, N_{3,4} \end{bmatrix} \begin{bmatrix} \mathbf{q}_{i-1,j-1} & \mathbf{q}_{i-1,j} & \mathbf{q}_{i-1,j+1} & \mathbf{q}_{i-1,j+2} \\ \mathbf{q}_{i,j-1} & \mathbf{q}_{i,j} & \mathbf{q}_{i,j+1} & \mathbf{q}_{i,j+2} \\ \mathbf{q}_{i+1,j-1} & \mathbf{q}_{i+1,j} & \mathbf{q}_{i+1,j+1} & \mathbf{q}_{i+1,j+2} \\ \mathbf{q}_{i+2,j-1} & \mathbf{q}_{i+2,j} & \mathbf{q}_{i+2,j+1} & \mathbf{q}_{i+2,j+2} \end{bmatrix} \begin{bmatrix} N_{0,4}(w) \\ N_{1,4}(w) \\ N_{2,4}(w) \\ N_{3,4}(w) \end{bmatrix}$$

Although a non-uniform B-Spline surface is a tensor product, sometimes there is an advantage in thinking of the surface as being made up of a collection of isoparametric curves. An important characteristic of non-uniform B-Spline surfaces is that a single knot sequence must be used for all of the isoparametric curves in each parametric direction. The blending functions for each parametric direction of non-uniform B-Spline surfaces are calculated recursively in the same manner as those in the B-Spline curve

formulation. However, if the surface is thought of as a series of isoparametric curves, all curves in one direction are being calculated using the same knot sequence. As a result, the representation of a grid of surface data in which the distance between the data points does not vary in the same pattern along each isoparametric curve using a non-uniform B-Spline surface can be difficult [Fari90b].

3.3 Discrete B-Splines

Discrete B-Splines have been utilized for several purposes ranging from solving minimization problems to iteratively computing non-linear splines [Cohe80]. Discrete B-Splines are defined over a knot sequence using a recursive algorithm. The properties of discrete B-Splines are similar to those of B-Splines discussed in the preceding sections of this chapter. In this thesis, discrete B-Splines are used to calculate the elements, $\alpha_{i,k}$, of the B-Spline knot insertion matrix described in the section of the literature survey on knot insertion. Recall that the B-Spline knot insertion matrix is used to transform the original control vertices, $\mathbf{V}_i$, into refined control vertices, $\mathbf{W}_i$. This can be expressed as:

$$\mathbf{W}_j = \sum_{i=0}^{m} \alpha_{i,k}(j)\mathbf{V}_i \quad for\, j = 0,...m+n$$

In a similar manner, a transformation from the original set of blending functions, $B_{i,k}$, to the refined set of blending functions, $N_{j,k}$, is expressed as:

$$B_{i,k}(t) = \sum_{j=0}^{m+n} \alpha_{i,k}(j) N_{j,k}(t).$$

Accordingly, the definition of a discrete B-Spline, $\alpha_{i,k}(j)$, is shown below:

$$\alpha_{i,1}(j) = \left\{ \begin{array}{ll} 1 & \text{if } t_i \leq \tau < t_{i+1} \\ 0 & \text{otherwise} \end{array} \right\}$$

$$\alpha_{i,r}(j) = \frac{(\tau_{j+r-1} - t_i)\,\alpha_{i,r-1}(j)}{t_{i+r-1} - t_i} + \frac{(t_{i+r} - \tau_{j+r-1})\,\alpha_{i+1,r-1}(j)}{t_{i+r} - t_{i+1}}$$

for $r = 2, 3, \ldots, k$, where k is the order of the B-Spline into which knots are being inserted.

The properties of discrete B-splines can be summarized as follows:

- Discrete B-Splines over a knot sequence always sum to one. That is, $\sum_{i=0}^{m} \alpha_{i,k}(j) = 1$.
- Discrete B-Splines have local support. That is, for any given j let δ be such that $t_\delta \leq \tau_j < t_{\delta+1}$, then $\alpha_{i,k}(j) = 0$ for $i \notin \{\delta - k + 1, \ldots, \delta\}$, for $i = 0, \ldots, m$.
- Discrete B-Splines are always non-negative. That is, $\alpha_{i,k}(j) \geq 0$ for all i, j, k.

Additional information on discrete B-Splines may be found in [Bart87], [Cohe80], [Lych75], [Lych76], [Schu73] and [DeBo76].

4.0 Data Reduction for Cubic B-Spline Curves

4.1 Difficulties Associated with Knot Removal

Knot removal for B-Spline curves and surfaces is a complicated process. While it is easy to say that the goal of knot removal is simply to remove a knot from a non-uniform B-Spline curve or surface, a more sensible approach is to address this task in a series of steps. This chapter presents the difficulties associated with knot removal for B-Spline curves by breaking the process down into three steps. First, the context of knot removal for B-Spline curves is established. Then the topics of classification of knots for knot removal, removal of knots from a B-Spline curve and approximation of a B-Spline curve using data reduction are addressed. The extension of curve-based knot removal to B-Spline surfaces is discussed in a later section of this thesis and will not be addressed here.

4.2 Overview of the Knot Removal Process

Let f be a B-Spline curve defined using a control polygon, Q, a knot sequence, $\mathbf{t} = \{t_i\}_{i=1}^{m+k}$, and blending functions, $B_{i,k,t}$. Then there is a space, $\mathbf{S}_{k,t}$, such that:

$$\mathbf{S}_{k,t} = \text{span}\{B_{1,k,t}, \dots, B_{m,k,t}\}$$

where m is the number of control vertices defining Q and k is the order of the B-Spline. If a reduced knot sequence, $\mathrm{T} = \{\tau_i\}_{i=1}^{n+k} \in \{t_i\}_{i=1}^{m+k}$, and blending functions, $N_{i,k,\tau}$, can be defined, then a second subspace, $\mathbf{S}_{k,\tau}$, exists such that:

$$\mathbf{S}_{k,\tau} = \text{span}\{N_{1,k,\tau}, \dots, N_{n,k,\tau}\}$$

where n is the number of control vertices defining the reduced curve and $m - n$ is the number of knots which were removed from the original curve, f. Since the reduced knot sequence, $\mathrm{T} = \{\tau_i\}_{i=1}^{n+k}$, is a subsequence of the original knot sequence, $\mathbf{t} = \{t_i\}_{i=1}^{m+k}$:

$$\mathbf{S}_{k,\tau} \in \mathbf{S}_{k,t}$$

The goal of knot removal, then, is to find an approximate B-Spline curve, g, defined in the subspace $\mathbf{S}_{k,\tau}$ such that:

$$||f - g|| < \varepsilon$$

where ε is the allowable error for the approximation g.

In attaining this goal, several questions arise. For instance, what knots should be removed from the original knot sequence, $\mathbf{t} = \{t_i\}_{i=1}^{m+k}$, to get the reduced knot

sequence, $T = \{\tau_i\}_{i=1}^{n+k}$? In addition, how must the locations of the control vertices be changed for the difference between the original curve and the approximating curve, $f - g$, to be within the specified tolerance? Finally, how should the answers to the two preceding questions be integrated to yield an algorithm for data reduction? These questions are addressed in the remaining three sections of this chapter.

4.3 Classification of Knots for Removal

The first step in knot removal for a B-Spline curve is to classify the knots of the curve so that their significance can be assessed. The significance of a knot should be measurable in some quantitative way which will yield a real number. The method used here is based on a weighting method originally presented by Lyche and Morken [Lych88]. Once the significance of the knots defining a B-Spline curve has been determined, the least significant knots can be removed from the curve. This section discusses the calculation of the weights which are used to determine the significance of the knots defining a B-Spline curve.

A weight must be determined for each interior knot of the B-Spline curve from which knots are being removed. However, it is easier to understand the process of determining a weight if the removal of only one knot is considered. Accordingly, the goal is to find the significance of a knot, t_j, from the original knot sequence, $\mathbf{t}$. This can be done by assuming the knot, t_j, and the control point, q_{j+k-2}, were never in the original curve, f. Then the knot at t_j is reinserted into the curve giving an approximating curve, f'. The steps of going from the original curve to the reduced curve and back to the

approximating curve are shown in Figure 5 on page 35. Accordingly, a reduced knot sequence, T, and a control polygon, $\mathbf{c}$, are used to define a reduced curve, g, as follows:

$$\mathrm{T} \subset \mathbf{t} \quad \ni \quad t_j \notin \mathrm{T}$$

$$\mathbf{c} \subset \mathbf{Q} \quad \ni \quad q_{j+k-2} \notin \mathbf{c}$$

Thus, T is a subset of $\mathbf{t}$ such that t_j is not a member of T. For example, if the original knot sequence is:

$$\mathbf{t} = \{t_i\}_{i=1}^{m+k} = \{0, 1, 3, 7, 9, 12, 13\}$$

and the knot for which a weight is to be found is $t_j = 7$, then:

$$\mathrm{T} = \{\tau_i\}_{i=1}^{m-1+k} = \{0, 1, 3, 9, 12, 13\}$$

Thus, f is defined on the knot sequence $\{0, 1, 3, 7, 9, 12, 13\}$ and g is defined on the knot sequence $\{0, 1, 3, 9, 12, 13\}$. When the knot $t_j = 7$ is inserted into g, a new control polygon, $\mathbf{d}$, must be found. A weight can be defined as:

$$w_j = \min_{dist(\mathbf{c},\ \mathbf{d})} \| f' - g \|_{\mathbf{t}}$$

The determination of the weight, w_j, can now be formulated in matrix notation. Let the knot for which a weight is being found be:

$$z = t_j \in \mathbf{t}$$

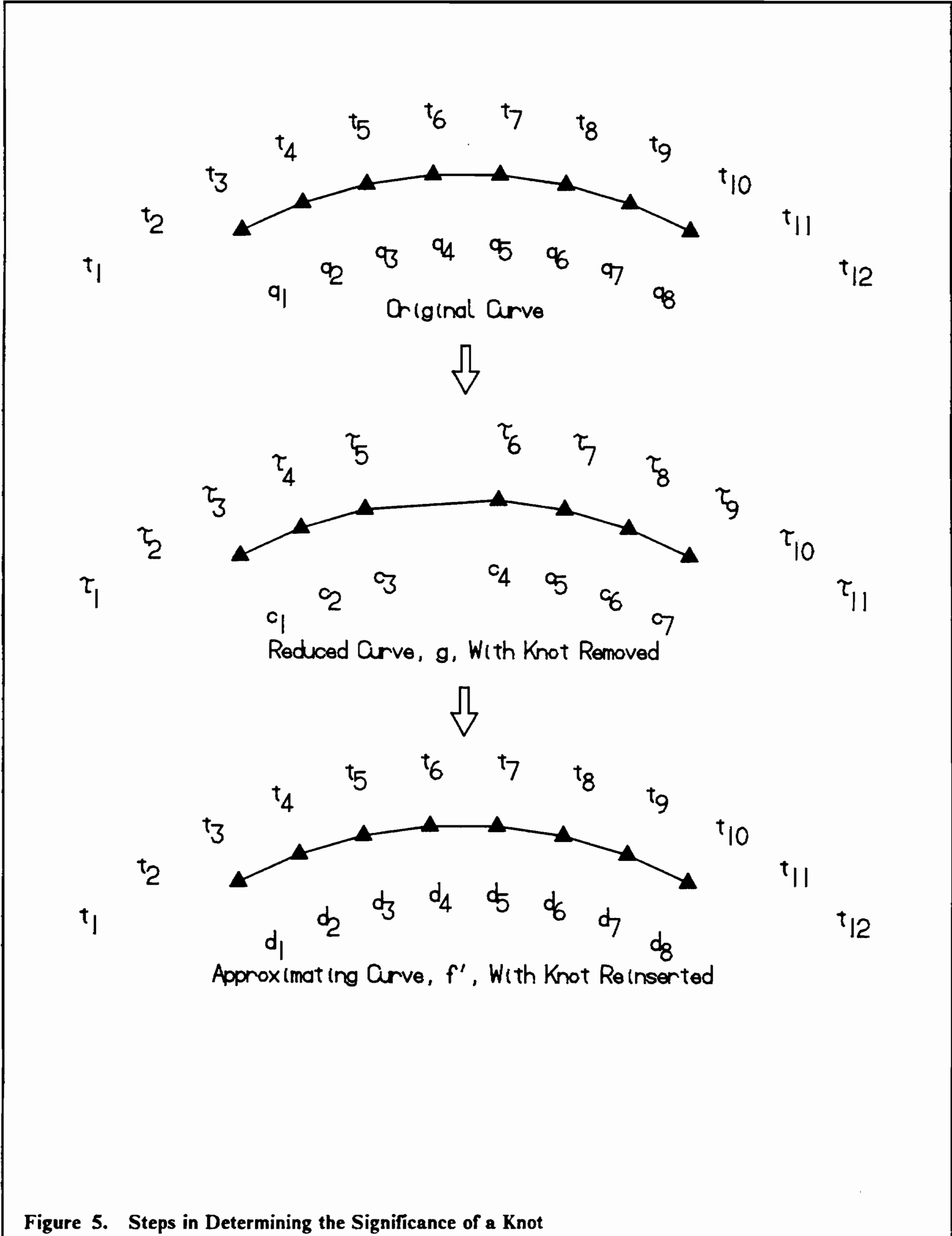


Figure 5. Steps in Determining the Significance of a Knot

so that $T \bigcup z = \mathbf{t}$. Therefore, $T \bigcup z$ is the knot sequence obtained by increasing the multiplicity of the knot z by one in the knot sequence T. Next, define integers v and l such that:

$$\tau_{v-l} < \tau_{v-l+1} = \ldots = \tau_{v-1} \leq z < \tau_v$$

where the terms $\tau_{v-l+1}, \ldots, \tau_{v-1}$ and z are multiple knots. The multiplicity of z in $\mathbf{t}$ is l and $v = j$, the subscript of the knot in $\mathbf{t}$ whose weight is being found.

The reduced curve defined on the knot sequence T can be defined as:

$$\mathbf{g} = \sum_{i=1}^{n} \mathbf{c}_i B_{i,k,\mathrm{T}} \in \mathbf{S}_{k,\mathrm{T}}$$

where the $\mathbf{c}_i$ are control vertices and $B_{i,k,\mathrm{T}}$ are blending functions. Likewise, the approximating curve takes the form:

$$\mathbf{f}' = \sum_{i=1}^{n+1} \mathbf{d}_i N_{i,k,\mathbf{t}} \in \mathbf{S}_{k,t}$$

where $\mathbf{d}_i$ are the control points and $N_{i,k,\mathbf{t}}$ are blending functions. The next step in formulating the matrix notation of $||f' - g||_t$ is to consider the process of inserting a knot into g to get f'. From Boehm's work on knot insertion [Boeh80]:

$$\begin{aligned}
d_i^0 &= c_i^0 && \mathit{for}\ i = 1, \ldots, v-k, \\
d_i^1 &= \mu_i c_i^1 + \lambda_i c_i^1 && \mathit{for}\ i = v-k+1, \ldots, v-l, \\
d_i^2 &= \overset{2}{c}_{i-1} && \mathit{for}\ i = v-l+1, \ldots, n+1
\end{aligned}$$

where

$$\lambda_i = \frac{\tau_{i+k-1} - z}{\tau_{i+k-1} - \tau_i} \quad \wedge \quad \mu_i = \frac{z - \tau_i}{\tau_{i,k-1} - \tau_i}$$

The second equation for transforming the $\mathbf{d}_i$ can be expressed in matrix form as:

$$\mathbf{d}^1 = \mathbf{B}\mathbf{c}^1$$

where $\mathbf{B}$ is a transformation matrix from $\mathbf{c}^1$ to $\mathbf{d}^1$. The $\mathbf{B}$ matrix takes the form:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ \lambda_{v-k+1} & \mu_{v-k+1} & \cdots & 0 & 0 \\ 0 & \lambda_{v-k+2} & \cdots & \vdots & \vdots \\ \vdots & \vdots & \cdots & \mu_{v-l-1} & 0 \\ 0 & 0 & \cdots & \lambda_{v-l} & \mu_{v-l} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

Thus, $\mathbf{B}$ has $k-l+2$ rows and $k-l+1$ columns. The elements μ_i decrease with increasing i, while the elements λ_i increase with increasing i. The control vertices of the approximating curve, f', can be found by solving the ℓ^∞ problem:

$$\min_c \; || \mathbf{d}^1 - \mathbf{B}\mathbf{c}^1 ||$$

where the ℓ^∞ problem consists of using the max norm to perform the above minimization.

The weight for knot t_j in the original knot sequence, $\mathbf{t}$, is set equal to:

$$w_j = || \mathbf{d}^l - \mathbf{B}\mathbf{c}^l || + \begin{Bmatrix} 0 & for\ l = 1 \\ w_{j-1} & for\ l > 1 \end{Bmatrix}$$

Since $\min_c || \mathbf{d}^l - \mathbf{B}\mathbf{c}^l ||$ is an overdefined system of equations, a special algorithm must be used to find a solution, **c**. Lyche and Morken developed the algorithm described below to solve such a set of equations.

In [Lych88], a linear system of equations is used to solve the minimization problem presented above. This solution takes the form:

$$\mathbf{A}\mathbf{c}^l = \mathbf{d}^l$$

where

$$\mathbf{A} = \begin{bmatrix} \mu_1 & 0 & \dots & 0 & 0 & (-1)^{n-1} \\ \lambda_2 & \mu_2 & \dots & 0 & 0 & (-1)^{n-2} \\ 0 & \lambda_3 & \dots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \dots & \mu_{n-2} & 0 & 1 \\ 0 & 0 & \dots & \lambda_{n-1} & \mu_{n-1} & -1 \\ 0 & 0 & \dots & 0 & \lambda_n & 1 \end{bmatrix}$$

and

$$n = k - l + 2$$

Accordingly, the **A** matrix is just the transformation matrix, **B** with an added column, **a**, defined by:

$$\mathbf{a} = \{ a_1, \ \dots, \ a_n \} \quad \ni \quad a_i = (-1)^{n-1}$$

With the elements of the **A** matrix defined, the following algorithm can be used to solve for $\mathbf{c}^1$:

1. Construct the matrix **A**.

2. Determine the largest integer p such that $\mu_p \geq 1/2$.

3. Eliminate $\mathbf{c}^1_{i-1}$ from equation i for $i = 2, \dots, p$ using forward Gaussian elimination on equations $2, \dots, p$.

4. Eliminate $\mathbf{c}^1_i$ from equation i for $i = n-1, \dots, p+1$ using backward Gaussian elimination on equations $n-1, \ n-2, \dots, p+1$.

5. Solve reduced equations p and $p+1$ for $\mathbf{c}^1_p$ and $\mathbf{c}^1_n$.

6. Perform back substitution on equation i for $i = p-1, \ p-2, \dots, 1$ to determine $\mathbf{c}^1_i$.

7. Perform back substitution on equation i for $i = p+2, \dots, n$ to determine $\mathbf{c}^1_{i-1}$.

Lyche and Morken [Lych88] have shown that this algorithm never results in division by zero, does not create fill-in and prevents growth of the elements in **A**, except for the last column. Therefore, the algorithm is stable.

When this process is carried out for the entire B-Spline curve, each interior knot is assigned a knot removal weight using:

$$w_j = \| \mathbf{d}^l - \mathbf{B}\mathbf{c}^l \| + \begin{Bmatrix} 0 & for\ l = 1 \\ w_{j-1} & for\ l > 1 \end{Bmatrix}$$

where the max norm is used. As of yet, the condition for multiple knots has not been implemented. Using these weights, the interior knots of the B-Spline curve can be sorted so that the least significant knot is at one end of a list and the most significant knot is at the other end of the list. A parallel quick sort is used for this purpose. The sorted list of knots and weights is called the knot removal list.

4.4 Calculation of New Control Vertices

The second step in removing knots from a B-Spline curve is to determine the new locations of control vertices in the vicinity of the knot being removed from the curve. The new locations for the remaining control vertices should be such that the amount the curve changes is minimal. This implies that the distance between the approximating curve, g, and the original curve, f, should be minimized. However, this is a difficult problem to solve. Accordingly, the task of finding the new locations for local control vertices while removing only one knot will be explained first.

Before solving the minimization problem, it is necessary to determine which knots and control vertices in a B-Spline curve must be included in the minimization if only one interior knot is being removed. The number of knots and control vertices which must be included in the minimization is dependent on the order of the curve. Assume knot t_j is to be removed from a cubic B-Spline curve (order $k = 4$). Then the control vertex

which is also being removed from the curve is $\mathbf{q}_{j-2}$. A local fourth order B-Spline which is defined by $k+1$ knots on either side of t_j and $k-1$ control vertices on either side of $\mathbf{q}_{j-2}$ must be considered in the minimization. This local region of the curve must be considered in the minimization problem because the curve segments defined by this region are affected by the knot t_j which is being removed. This is shown in Figure 6 on page 42.

For a minimization problem of this type, an ℓ_p norm $(1 \le p \le \infty)$ is typically used. The norms used in this thesis are defined over any knot sequence, $\mathbf{t}$, containing a reduced knot sequence, T, as:

$$||f|| = \begin{cases} \sum_j |\mathbf{d}_j|^p [(t_{j+k} - t_j)/k]^{1/p} & \text{for } 1 \le p < \infty, \\ \max_j |\mathbf{d}_j| & \text{for } p = \infty \end{cases}$$

where the $\mathbf{d}_j$ and t_j are the control vertices and knots, respectively, of the original curve, $f \in \mathbf{S}_{k,\mathbf{t}}$. If $\mathbf{c}_j$ are the control vertices of the approximating curve, $g \in \mathbf{S}_{k,\mathrm{T}}$, then $\mathbf{d} = \mathbf{B}\mathbf{c}$ where $\mathbf{B}$ is the B-Spline knot insertion matrix of order k from T to $\mathbf{t}$. Therefore, the above equation can be written:

$$||f||_{\ell^p,\mathbf{t}} = ||\,\mathbf{E}_{\mathbf{t}}^{1/p}\,\mathbf{B}\mathbf{c}\,||_p$$

where $\mathbf{E}_{\mathbf{t}}^{1/p}$ is an $m \times m$ scaling matrix defined by:

$$e_{i,i} = \begin{cases} [(t_{i+k} - t_i)/k]^{1/p} & \text{for } 1 \le p < \infty, \\ 1 & \text{for } p = \infty \end{cases}$$

Lyche and Morken [Lych87] discussed the use of discrete norms for approximating B-Spline curves. However, the decision of which ℓ_p norm to use was not answered.

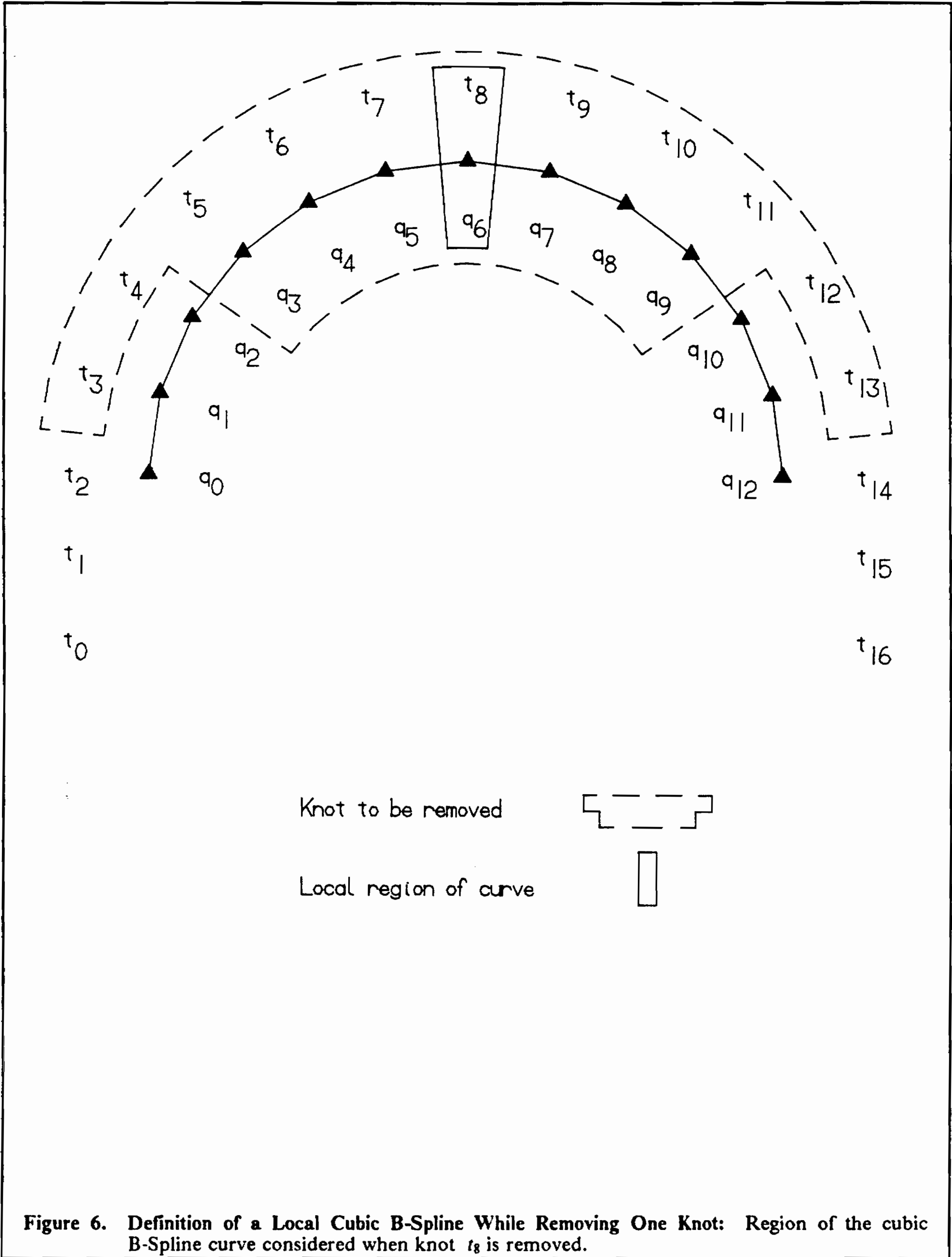


Figure 6. Definition of a Local Cubic B-Spline While Removing One Knot: Region of the cubic B-Spline curve considered when knot t_8 is removed.

Throughout the literature, it has been well documented that the control polygon of a B-Spline curve converges to the curve as the control polygon is refined [Cohe80], [Bart87]. This implies that the $(\ell_\infty, \mathbf{t})$ norm converges to the $\mathbf{L}_\infty$ norm of the B-Spline. In a later paper, Lyche and Morken [Lych88] have shown that the difference of the $(\ell_2, \mathbf{t})$ norm and the $\mathbf{L}_2$ norm is small if the curve is smooth. Therefore, this norm is used to calculate the new locations of the control vertices near the knot which is being removed. The new control vertices will be the solution to the least squares problem:

$$\min_{g \in \mathbf{S}_{k,\tau}} \| f - g \|_{\ell_{2,t}}$$

Using the B-Spline knot insertion matrix and the scaling matrix, the least squares problem can be expressed as:

$$\min \| \mathbf{E}_t^{1/2}(\mathbf{Bc} - \mathbf{d}) \|$$

In a least squares solution of over determined linear systems, ill-conditioned matrices are sometimes a concern. Lyche and Morken [Lych88] have shown that for this system, if a suitable scaling matrix is introduced, the condition number of the coefficient matrix is bounded by D_k^2 [DeBo76] independently of the knot sequences. The scaling matrix which must be introduced is $\mathbf{E}_T^{1/2}$. This $n \times n$ scaling matrix is defined similar to $\mathbf{E}_t^{1/2}$:

$$e_{i,i} = [(\tau_{i+k} - \tau_i)]^{-1/2}$$

When this scaling matrix is introduced, the least squares minimization becomes:

$$\min_{x \in \mathbf{R}^n} \| \mathbf{E}_t^{1/2}(\mathbf{B}\mathbf{E}_T^{-1/2}\mathbf{x} - \mathbf{d}) \|$$

The new control points can be found by first solving for $\mathbf{x}$ then substituting into:

$$\mathbf{c} = \mathbf{E}_{\mathbf{T}}^{-1/2}\mathbf{x}$$

The least squares problem can be solved using one of two methods. The first method uses orthogonal transformations. The second method utilizes normal equations to solve the least squares problem. This approach is safe to use because the condition number of the coefficient matrix is bounded by D_k^2 [DeBo76]. For a fourth order B-Spline curve ($k = 4$), $D_k = 10.03$.

Both the method of orthogonal transformations and the utilization of normal equations were implemented during the development of software for this thesis. Since these techniques gave identical results, normal equations are used to solve the least squares minimization in the final version of this thesis. If higher order B-Spline curves are used, the solution of this least squares minimization using normal equations and orthogonal projections should be checked. Values of D_k for B-Spline curves of various orders are listed in [DeBo76].

Recall that the discussion so far in this section holds only for removing one knot from a cubic B-Spline curve (order $k = 4$). However, this can be extended to removing more than one knot, especially if the knots to be removed are all located within the same local curve as defined in Figure 6 on page 42. If a list of several nearby knots is to be removed, then the local curve which must be considered will include one or more additional knots and control vertices. In this case, the region of the B-Spline curve which must be considered can be expressed in terms of the highest and lowest subscripts of the knots which are in the removal list. The local cubic B-Spline curve ($k = 4$) which must be considered in removing a list of knots is defined by the knots from $t_{low-k-1}$ to t_{hi+k+1} and by the control vertices from $\mathbf{q}_{low-k+1}$ to $\mathbf{q}_{hi+k-1}$. The subscripts *hi* and *low*

are the subscripts of the knots in the removal list having the highest and lowest values, respectively. Two examples are given in Figure 7 on page 46 and Figure 8 on page 47. Again, this local region of the curve must be considered in the minimization problem because the curve segments defined by this region are affected by the knots which are in the removal list. If two or more knots are to be removed from a cubic B-Spline curve and they are far enough apart that they do not affect the same local region of the curve, then they must be removed separately. This is shown in Figure 9 on page 48.

4.5 A Data Reduction Algorithm for B-Spline Curves

An algorithm for data reduction must incorporate the previous two steps which have been described in this chapter, namely, determining which knots to remove and then removing them. The ideal algorithm for data reduction on a B-Spline curve would be:

1. Save the initial or reduced representation of the curve as the current representation of the curve.

2. Determine the knot which has the least significance.

3. Remove the knot having the least significance from the curve.

4. Check whether the reduced curve is within the specified tolerance.

5. If the reduced curve is within the specified tolerance, then repeat steps one through four. Otherwise, go to step six.

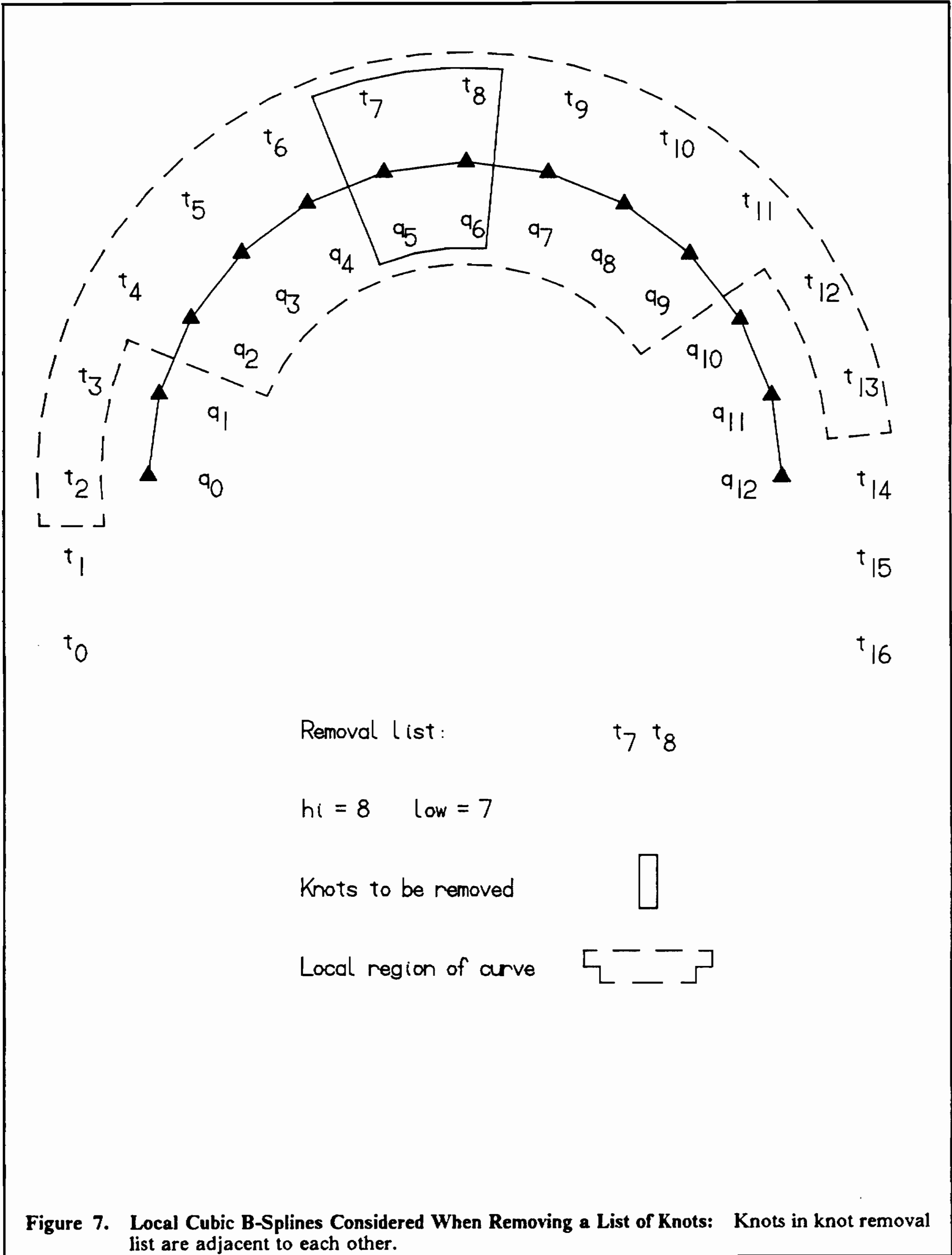


Figure 7. Local Cubic B-Splines Considered When Removing a List of Knots: Knots in knot removal list are adjacent to each other.

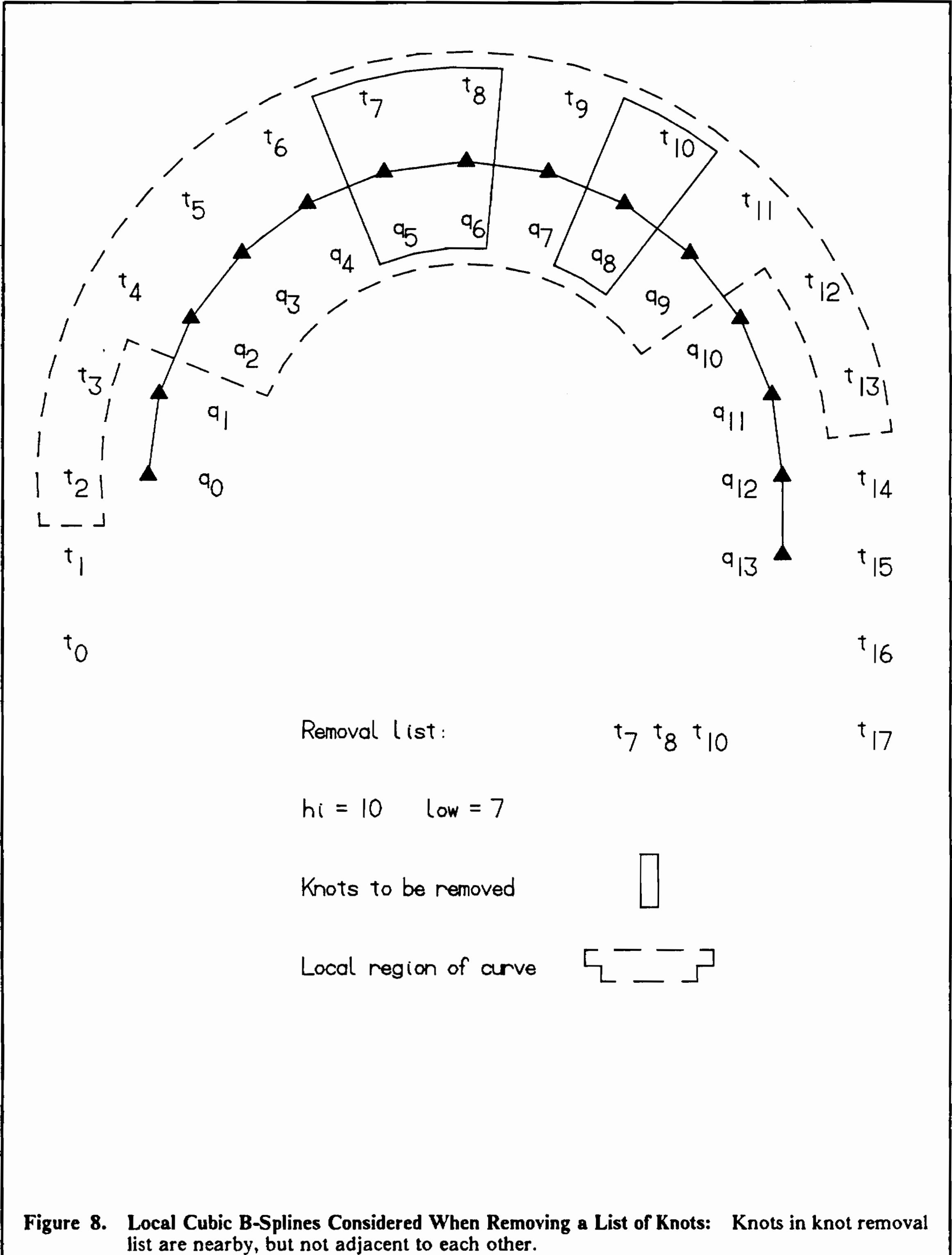


Figure 8. Local Cubic B-Splines Considered When Removing a List of Knots: Knots in knot removal list are nearby, but not adjacent to each other.

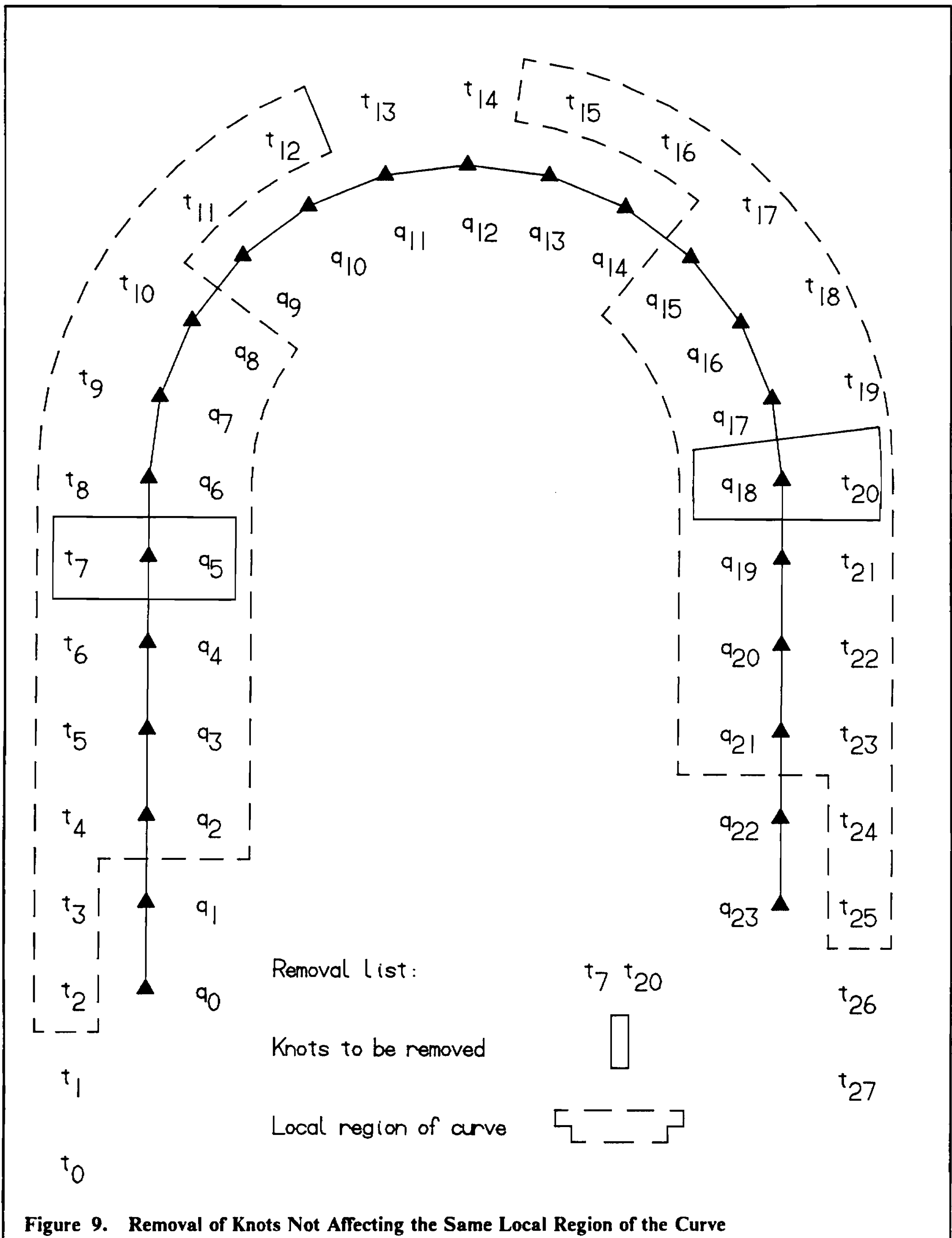


Figure 9. Removal of Knots Not Affecting the Same Local Region of the Curve

6. Replace the initial representation of the curve with the current representation.

This algorithm seems to be good since it removes knots from the B-Spline curve in a prioritized manner until the tolerance is exceeded and then gives the best approximation within the tolerance. However, the calculations necessary to find the weights for each new curve representation in step two are computationally intensive. For a curve having on the order of 10^2 or more knots, the above knot removal algorithm is relatively slow. To increase the speed, knots can be removed in localized groups or partitions. If the partition size is chosen wisely, knots can still be removed in a prioritized manner, however, more knots can be removed during each iteration.

Two partitioning methods were implemented during the development of software for this thesis. The first consisted of an adaptive partitioning method based on the tolerance, ε, the allowable difference between the original B-Spline curve and the approximating B-Spline curve. The second partitioning method consisted of fixing the number of knots being removed during each iteration and adjusting the number of knots removed in the final iteration so that the desired number of knots were removed from the curve. Both partitioning methods are described below.

The adaptive partitioning method is based on work done by Lyche and Morken [Lych88]. The tolerance, ε, is used to calculate values which are used to partition the knots based on knot removal weights. The partitions are:

$$0 = p_0 < p_1 < \dots < p_q = w_{j_{\max}}$$

Here, $w_{j_{\max}}$ is the maximum weight in the list of knot removal weights calculated for the interior knots of a curve. The partitioning values, p_i, are calculated using:

$$p_i = \begin{cases} 0 & for\ i = 0 \\ 2^{i-2}\varepsilon & for\ i = 1, \dots, q \end{cases}$$

where q is the number of partitions needed to partition the entire list of knot removal weights.

Starting at partition p_0, knots are removed until the desired amount of data reduction is achieved. However, knot removal weights and partitions are not recalculated for the intermediate B-Spline curves. This could result in the least significant knots failing to be removed from the curve during the removal of subsequent partitions. In fact, the partition p_0 was always found to contain no knots, while the partition p_1 was found to usually contain only one knot. Therefore, even if knot removal weights were calculated and the knots were re-partitioned between subsequent iterations, this method would be no faster than the algorithm suggested at the beginning of this section. Accordingly, this partitioning method is not used in the final implementation of the data reduction software.

The fixed partitioning method, which is used in the final implementation of this thesis, removes five knots from the curve during each iteration until the desired number of knots have been removed from the curve. Knot removal weights are recalculated after each partition of five knots is removed. This decreases the number of times that weights must be calculated compared to the adaptive method described above. This algorithm takes advantage of the fact that knots having the least significance are often near each other. If two or more of the five knots being removed are found to be within the same local region of the curve, then these knots are removed at the same time. Otherwise, the knots in the partition are removed one at a time. The removal of knots which are near

each other is done using sub-partitions. For example, if the original partition of five knots which is to be removed consists of $\{ t_7, t_{35}, t_{23}, t_8 , t_{11} \}$ then three sub-partitions are created:

$$\{ t_7, t_8, t_{11} \} \quad \{ t_{35} \} \quad \{ t_{23} \}$$

The knots in each of these sub-partitions are removed using the least squares method previously presented. After removing five knots or a full partition (if the partition size is less than five knots), knot removal weights are recalculated for the reduced representation of the B-Spline curve. This process is repeated until the desired number of knots have been removed from the curve.

The data reduction algorithm for B-Spline curves using fixed partitioning can be summarized as follows:

1. Save the initial or reduced representation of the curve as the current representation of the curve.

2. Determine which knots in the current representation of the curve have the least significance.

3. Determine the number of knots to be removed during the current iteration (≤ 5) and make an appropriate partition.

4. Divide the above knot partition into ordered sub-partitions based on whether or not any of the knots are in the same local regions of the current curve representation.

5. Remove the sub-partitions from the current representation of the curve in prioritized order.

6. Check to see if enough knots have been removed from the initial representation of the curve.

7. If the data is reduced by the desired amount then go to step eight. Otherwise, repeat steps one through six.

8. Replace the initial representation of the curve with the final reduced representation.

5.0 Data Reduction for Bi-cubic B-Spline Surfaces

Data reduction and knot removal for bi-cubic B-Spline surfaces is a direct extension of the algorithm used for non-uniform B-Spline curves. However, instead of removing individual knots, an entire cross section of knots must be removed from the control polyhedron defining the B-Spline surface. The algorithm for curves is applied first in one parametric direction, and then in the other parametric direction. If it is desirable to reduce the data in only one parametric direction, then this can be done simply by appling curve reduction in that direction. This chapter describes classification of knots for removal of from a bi-cubic B-Spline surface, removal of knots from a bi-cubic B-Spline surface, and the integration of the two preceeding tasks into a data reduction algorithm.

5.1 Classification of Knots for Removal from a B-Spline Surface

The classification of knots for removal from a B-Spline surface is done by repeating the algorithm for determining the significance of knots for a curve. In each parametric direction, the knots on curves which define a cross section are assigned weights. Then the weights for equivalent knots on each cross section of the surface are averaged to get a weight for that row of knots. The weights calculated for the rows of knots and row numbers are then sorted using a parallel quick sort. The rows of knots having the lowest weights are removed first.

5.2 Calculation of New Control Vertices

Control vertices for the reduced bi-cubic B-Spline surface are also found by using the curve-based least squares algorithm on a cross section by cross section basis. To perform data reduction in both parametric directions, new control vertices for isoparametric curves in one direction are calculated. Then the new control vertices for isoparametric curves in the other direction are found. For each isoparametric curve, the new control vertices are located in the same "isoparametric plane" as the control points of the original isoparametric curve. Thus, the surface representation only changes in the parametric direction in which the isoparametric curves are being reduced.

5.3 Data Reduction for B-Spline Surfaces

The algorithm for data reduction of non-uniform B-Spline surfaces consists of repeating curve-based data reduction for each isoparametric curve defined by the rows or columns of vertices in the control polyhedron. Thus, to reduce a surface in both parametric directions, the surface is first reduced in one direction and then in the other direction. The algorithm for reducing a non-uniform B-Spline surface in one parametric direction is presented below.

1. Determine the significance of the interior knots along each isoparametric curve by calculating their knot removal weights.

2. Average the knot removal weights of each row of knots along the surface to determine the significance of each row of knots in the surface representation.

3. Sort the averaged knot removal weights to find the best rows of knots to remove from the surface.

4. Remove the rows of knots having the lowest average weights using the least squares minimization on each isoparametric curve of the surface.

6.0 Integration of Data Reduction into the ACSYNT B-Spline Module

The data reduction algorithms for curves and surfaces which have been presented in this thesis were implemented in the ACSYNT B-Spline module using the C programming language and graPHIGS. When ever possible, dynamic memory allocation and existing software modules were used. Appendix A contains a source code listing of modules which were added to the B-Spline module during the implementation of data reduction for non-uniform B-Spline curves and surfaces.

The original data structures used to store information associated with a model and its components were modified during the implementation of data reduction. Previous and next model pointers were added to the model data structure to allow a doubly linked list of models to be used. These pointers provide a method for storing reduced models and viewing both the non-reduced and reduced models.

The ability to manipulate models was also added to the user interface. This allows the user to specify the model to be displayed, the names of different models, and whether to move to the previous or next model in the linked list.

Three new items were added to the component data structure. The first two are the number of knots removed from the component in the u and w directions. The third item which was added to the component data structure was a two dimensional array of pointers to structures containing information on the weights assigned to each knot. The structures pointed to by this two dimensional array were called removal data structures. The array consists of one array element for each knot defining the component surface. Each of these structures contain the control vertex index and the weight which is assigned to the knot associated with that control vertex during the classification step of data reduction. The modified model data structure is shown in Figure 10 on page 58, while the modified component data structure is described in Figure 11 on page 59. The removal data structure is shown in Figure 12 on page 60.

Two additional data structures are temporarily used during the data reduction algorithm. The first of these data structures, the weight data structure, is used during the classification of knots for removal from a non-uniform B-Spline curve or surface. A detailed description of this data structure is given in Figure 13 on page 61. A weight data structure is created and destroyed each time the knots on a curve are classified for removal. The second temporary data structure which is used during the data reduction algorithm is the approximation data structure. A description of the approximation data structure is given in Figure 14 on page 62. The approximation data structure is created and destroyed each time a partition of five knots is removed from a curve.

Model Data Structure

```
typedef struct {
        char model_name¢20!;              /name of the model */
        int num_comp;                     /* number of components in model */
        int acs_root;                     /* root structure id */
        int nubs_root;                    /* Non-Uniform B-Spline root id */
        int int_root;                     /* Structure id for intersection data */
        int fillet_root;
        comp_data *comp;                  /* pointer to beginning of linked list */
        struct intersection_type *intlist; /* list of intersections */
        MODEL_type *prev;                 /* pointer to previous model in linked list */
        MODEL_type *next;                 /* pointer to next model in linked list */
                                          }MODEL;
```

Figure 10. Modified Model Data Structure

Component Data Structure

```
typedef struct compdata_type {
        int comp_number;               /* component number */
        char comp_name[20];            /* component name */
        int acs_id;                    /* structure id     */
        int nubs_id;
        int *hull_id;
        int fillet_id;
        int open[2];                   /* open flag 1 closed 0 open */
        int color;                     /* component color */
        int existence;                 /* 1 exists 0 does not exist */
        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 pts */
        float ***acs_utan;             /* pointer to tangents in u dir */
        float ***acs_wtan;             /* pointer to tangents in w dir */
        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 */
        rem_data ***rem;               /* array of removal data structures */
        int nu_removed;                /* number of knots removed in u direction */
        int nw_removed;                /* number of knots removed in w direction */
        struct compdata_type *next;    /* pointer to next component */
                                       }comp_data;
```

Figure 11. Modified Component Data Structure

Removal Data Structure

```
typedef struct rem_data_type {
        int index;                    /* control point number */
        float weight;                 /* knot removal weight */
                                      }rem_data;
```

Figure 12. Removal Data Structure for Storing Weights

Weight Data Structure

```
typedef struct weight_data_type {
        int n_knots;            /* number of knots after knots are removed */
        int rem_index;          /* index of knot for which weight is being found */
        float rem_val;          /* knot value to be removed */
        int p;                  /* subscript of control vertex such that MU > = 1/2 */
        float *knot;            /* pointer to reduced knot vector */
        float *hull;            /* pointer to reducte control hull */
        float *trans;           /* pointer to coefficient matrix used to transform
                                   old control vertices into new control vertices */
        float *init_trans;      /* pointer to the initial coefficient matrix,
                                   trans */
        float *rhsx;            /* x coordinates of intermediate control hull */
        float *rhsy;            /* y coordinates of intermediate control hull */
        float *rhsz;            /* z coordinates of intermediate control hull */
        float *init_rhsx;       /* x coordinates of intermediate control hull */
        float *init_rhsy;       /* y coordinates of intermediate control hull */
        float *init_rhsz;       /* z coordinates of intermediate control hull */
        float *solnx;           /* x coordinates of new control hull */
        float *solny;           /* y coordinates of new control hull */
        float *solnz;           /* z coordinates of new control hull */
        float weight;           /* weight assigned to knot rem_index in original
                                knot sequence and control hull */
                                }WEIGHT;
```

Figure 13. Weight Data Structure Used During Classification of Knots

Approximation Data Structure

```
typedef struct rem_data_type {
        int k;                  /* order of B-splines being used */
        int m;                  /* number of control points in original curve
        int n;                  /* number of control points in curve after knots
                                   are removed */
        int rem_index;          /* subscript of knot currently being
                                   removed in the range 1...m */
        int low_index;          /* lowest subscript (index) of the list being
                                removed */
        int hi_index;           /* highest subscript (index) of the list being
                                   removed */
        int num_to_be_removed;/* number of knots to be removed
                                   from original curve */
        int *removal_list;      /* list of knot subscripts (starting at
                                   zero) to be removed from original curve */
        float **d,              /* control vertices before knot removal */
        float *t;               /* original knot sequence */

        float **x;              /* control vertices after knot is removed */
        float *tau;             /* knot sequence after control point is removed */
        float **scale_t;        /* scaling matrix E(1/2) on original knot
                                   sequence, t */
        float **B;              /* B-spline knot insertion matrix for k=4 from
                                   the knot sequence tau to the knot sequence t */
        float **scale_tau;      /* scaling matrix E(-1/2) on reduced knot
                                   sequence, tau */
        float **bscale_tau;     /* product of (B) X (scale_tau) */
        float **scale_b_scale;  /* product of (scale_t) X (B) X (scale_tau) */
        float **scale_d;        /* product of (scale_t) X (d) */
        float **lhs_normal;     /* coefficients for unknowns in system of
                                   normal equations*/
        float **rhs_normal;     /* right hand side of the system of
                                   normal equations */
                                }APPROXIMATION;
```

Figure 14. Approximation Data Structure Used During Knot Removal

7.0 Results

7.1 Non-Uniform B-Spline Curve Reduction

Two examples of non-uniform B-Spline curve reduction are provided to illustrate the strengths and weaknesses of the data reduction algorithm. The first curve is a cross section of an airfoil which was originally defined using 104 knots and 100 control vertices. The airfoil has a chord length of 16 feet (4.877 meters) and a quarter chord thickness of 3 feet (0.914 meters). When the data reduction algorithm was used to remove knots from the curve, a 50 percent reduction was achieved, while introducing an error in the approximating curve on the order of 1.2×10^{-3} inches (3×10^{-5} meters). This seems to be quite reasonable. The original curve is shown in Figure 15 on page 64 and the reduced curve is shown in Figure 16 on page 65. While this example shows the strength of the algorithm for data reduction of non-uniform B-Spline curves, the second example illustrates the weakness of the algorithm. This curve, which is similar to the cross section of an I-beam, was chosen because it has varying curvature and several inflection points.

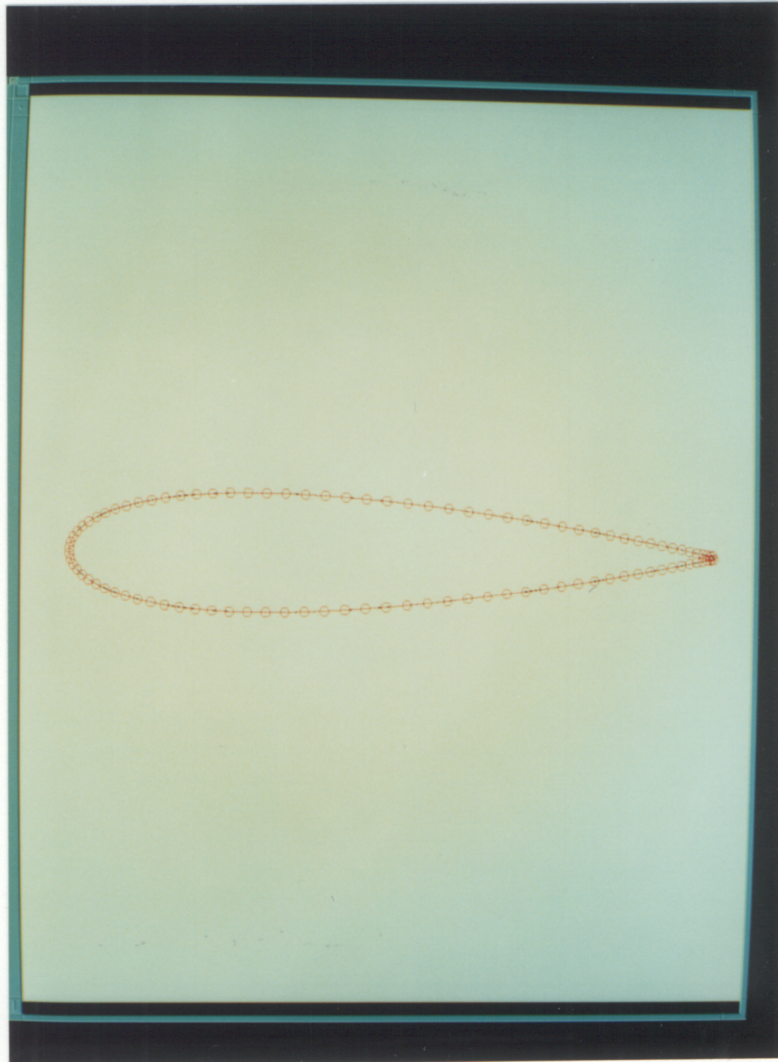


Figure 15. Original Airfoil Curve, 104 Knots and 100 Control Vertices

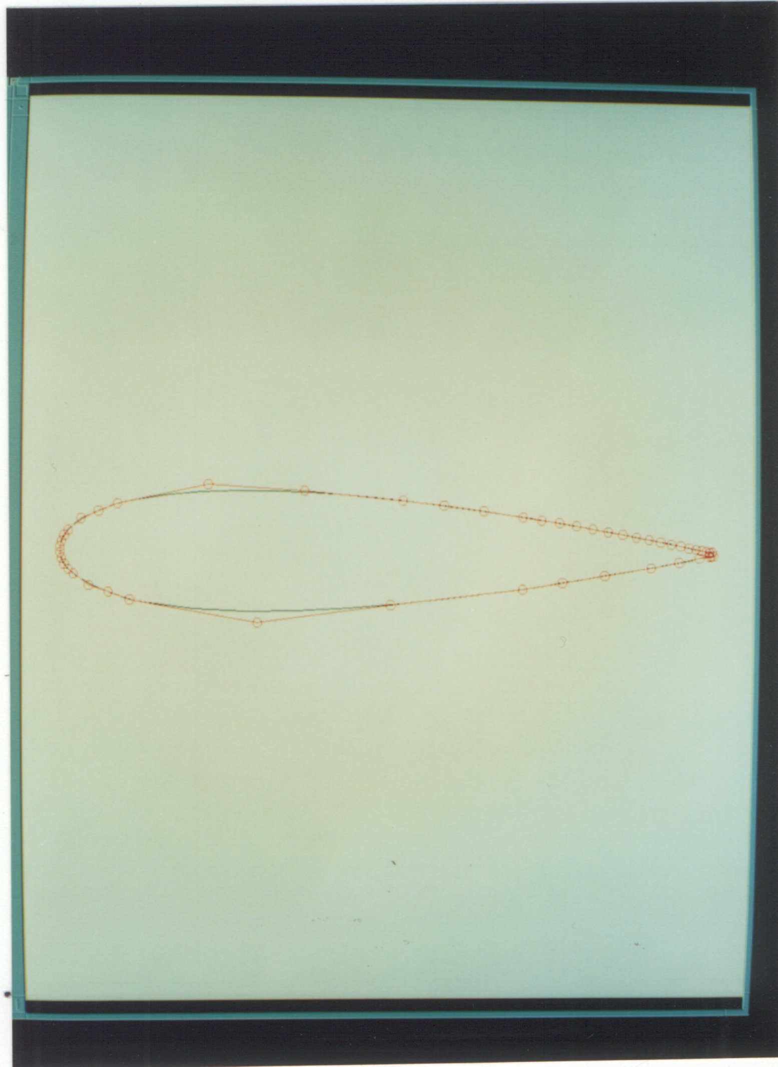


Figure 16. Reduced Airfoil Curve, 54 Knots and 50 Control Vertices

The curve is analytically defined as:

$$y = 5 + \frac{(x^3 - x)\sin(\pi x/180)}{7500} \qquad for \quad -180 \le x < 180$$

This function produces the curve shown in Figure 17 on page 67 which was represented using 405 knots and 401 control vertices. A data reduction of 20 percent was achieved using the data reduction algorithm. Thus 80 of the 405 knots were removed from the original curve. The reduced curve is shown in Figure 18 on page 68. Note that all of the knots which were removed, were located in the same region of the curve. Although it appears that additional knots could be removed, if more than 80 knots are removed, the classification step of the data reduction algorithm fails. Accordingly, the wrong knots are removed from the curve and the convexity of the curve is not retained. This is shown in Figure 19 on page 69 where 160 of the 405 knots were removed. The areas where knots were incorrectly chosen to be removed are along the lower flange. Methods for correcting this difficulty in choosing which knots to remove are discussed in the next chapter on recommendations. However, for the present, it suffices to simply observe that the best knots were not chosen to be removed from the curve. The knots which should have been removed are those along the portions of the curve having low curvature.

7.2 *Surface Reduction*

Two examples of non-uniform B-Spline surface reduction are presented. The first example is the parabolic surface shown in Figure 20 on page 70. This surface is defined

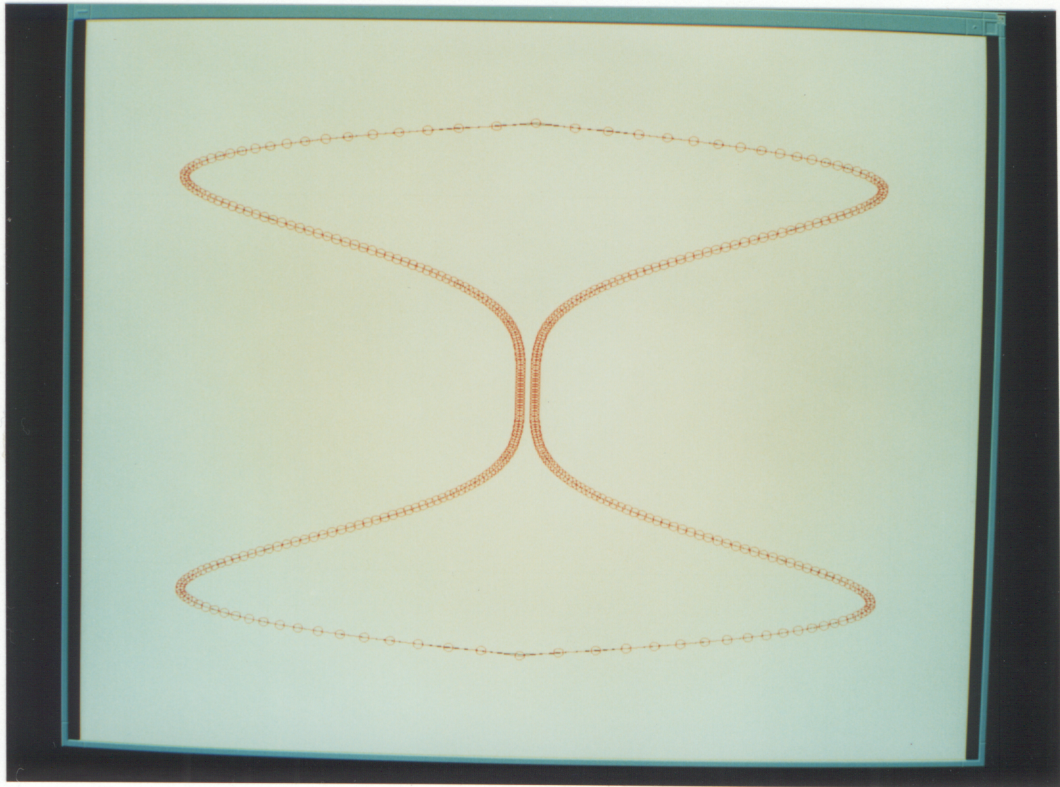


Figure 17. Original I-Beam Curve, 405 Knots and 401 Control Vertices

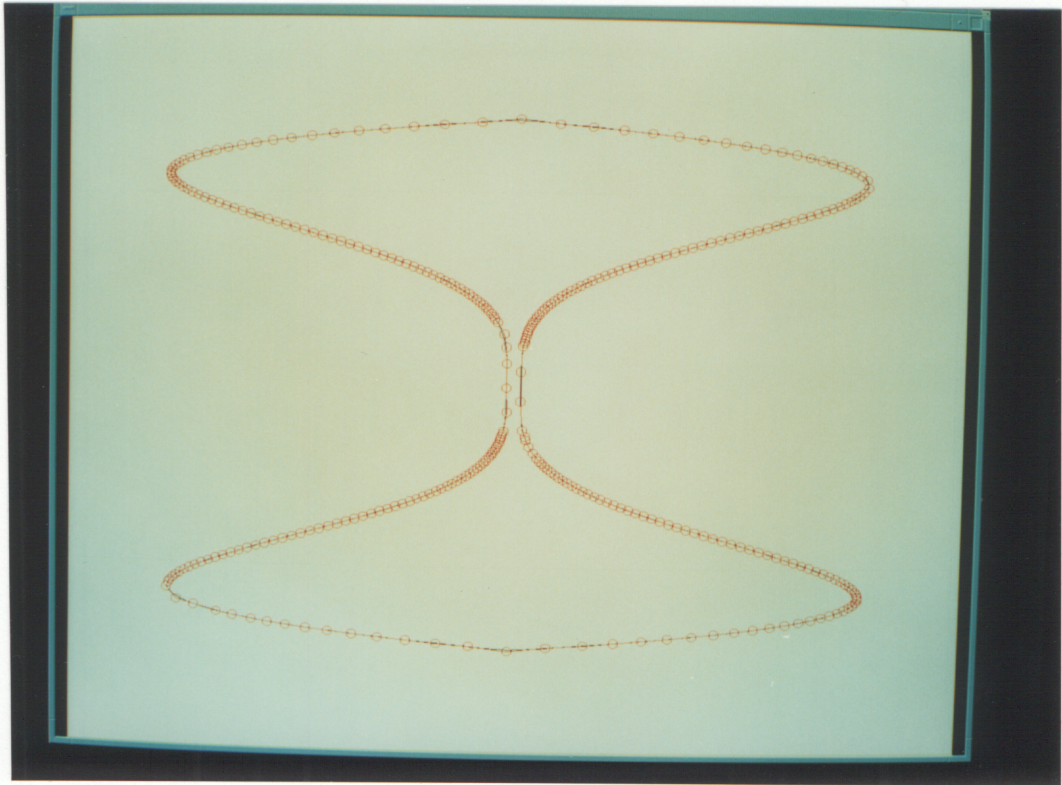


Figure 18. Reduced I-Beam Curve, 325 Knots and 321 Control Vertices

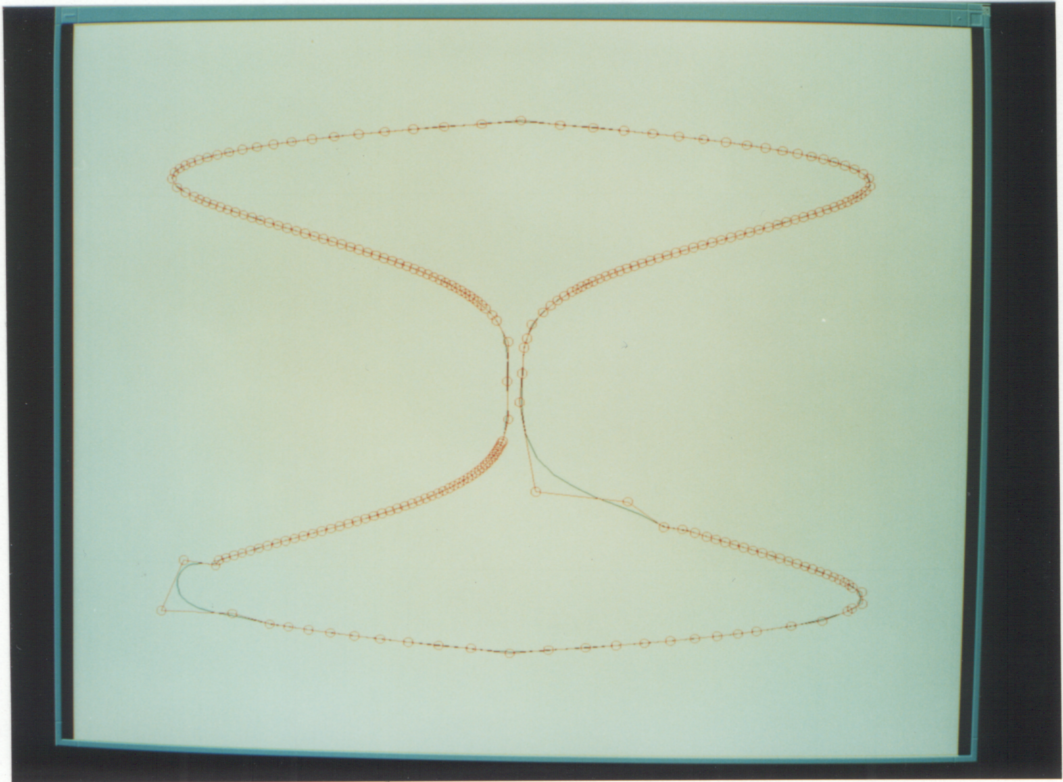


Figure 19. Reduced I-Beam Curve, 245 Knots and 241 Control Vertices

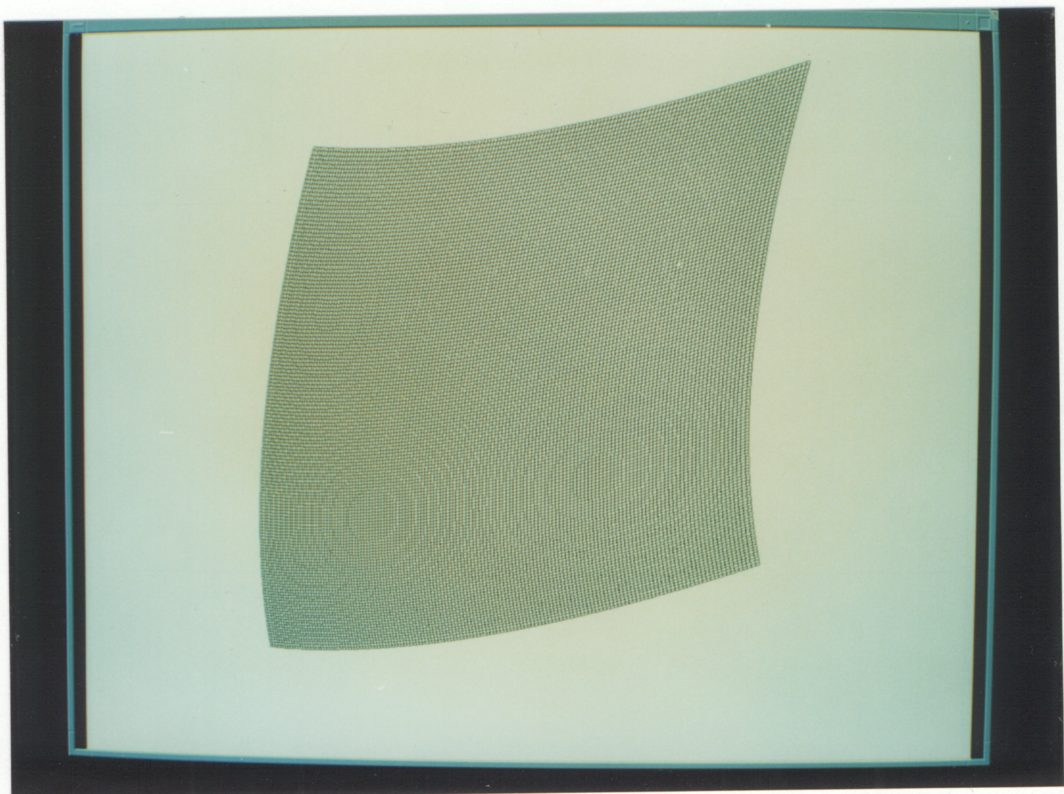


Figure 20. Original Parabolic Surface, 75 x 75 Vertex Control Polyhedron

by a mesh of 79×79 knots and 75×75 control vertices. The parabolic surface was reduced using curve reduction in each parametric direction. The surface was first reduced in the u parametric direction by removing 63 knots from each isoparametric curve. The result of this reduction, called the intermediate surface, is shown in Figure 21 on page 72. From Figure 21 on page 72, it can be concluded that reducing a surface in only one parametric direction does not affect the accuracy of the surface representation in the other direction. Next, the intermediate surface was reduced in the w parametric direction by removing 63 knots from each isoparametric curve in that direction. The result is the final reduced surface shown in Figure 22 on page 73 which is reduced by 70.5 percent when compared with the original surface. This extensive amount of data reduction is possible because of the lack of inflection points and large variations in curvature. Although an algorithm to determine the error between the original surface and the reduced surface was not implemented, when the reduced surface is densely tiled, no irregularities can be seen in the surface. The second example of non-uniform B-Spline surface reduction is for an aircraft wing. The original wing was defined using 25 cross sections. Each cross section was defined using 104 knots and 100 control vertices. Thus the wing was defined using a 25×100 vertex control polyhedron. This representation was reduced by 32 percent to a surface defined by a 16×50 vertex control polyhedron. The original wing is shown in Figure 23 on page 74, while the reduced wing is shown in Figure 24 on page 75. Again, no waves or new inflection points can be seen on the reduced wing when it is densely tiled.

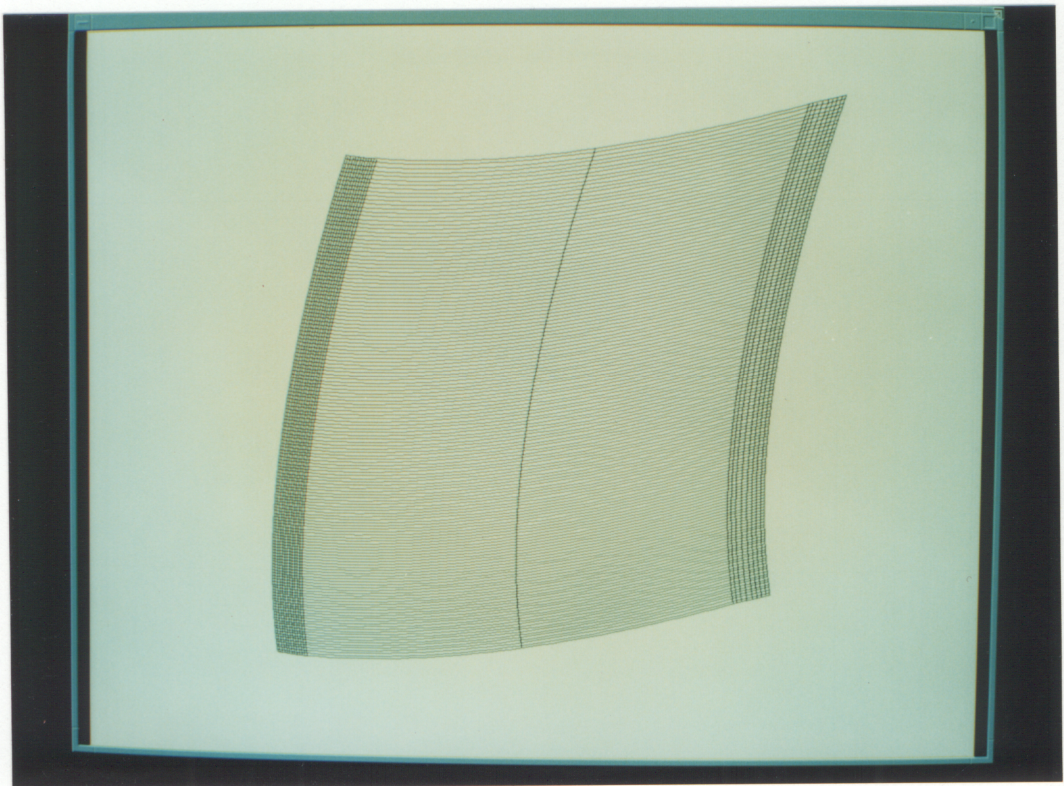


Figure 21. Intermediate Parabolic Surface, 12 x 75 Vertex Control Polyhedron

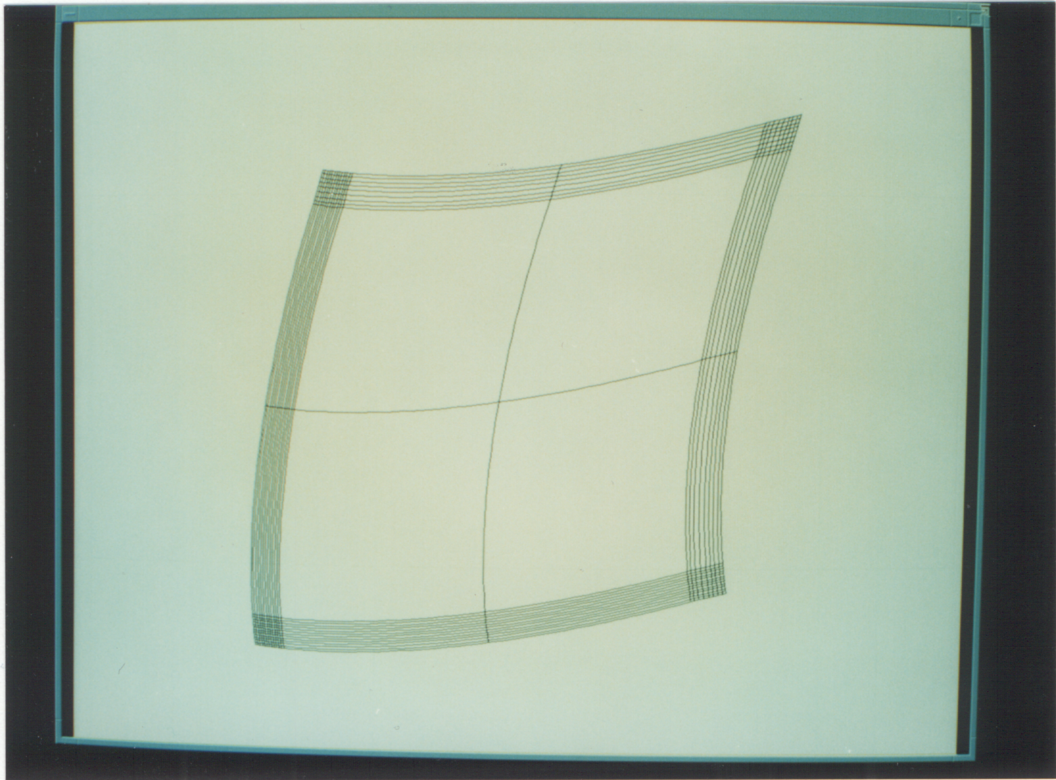


Figure 22. Reduced Parabolic Surface, 12 x 12 Vertex Control Polyhedron

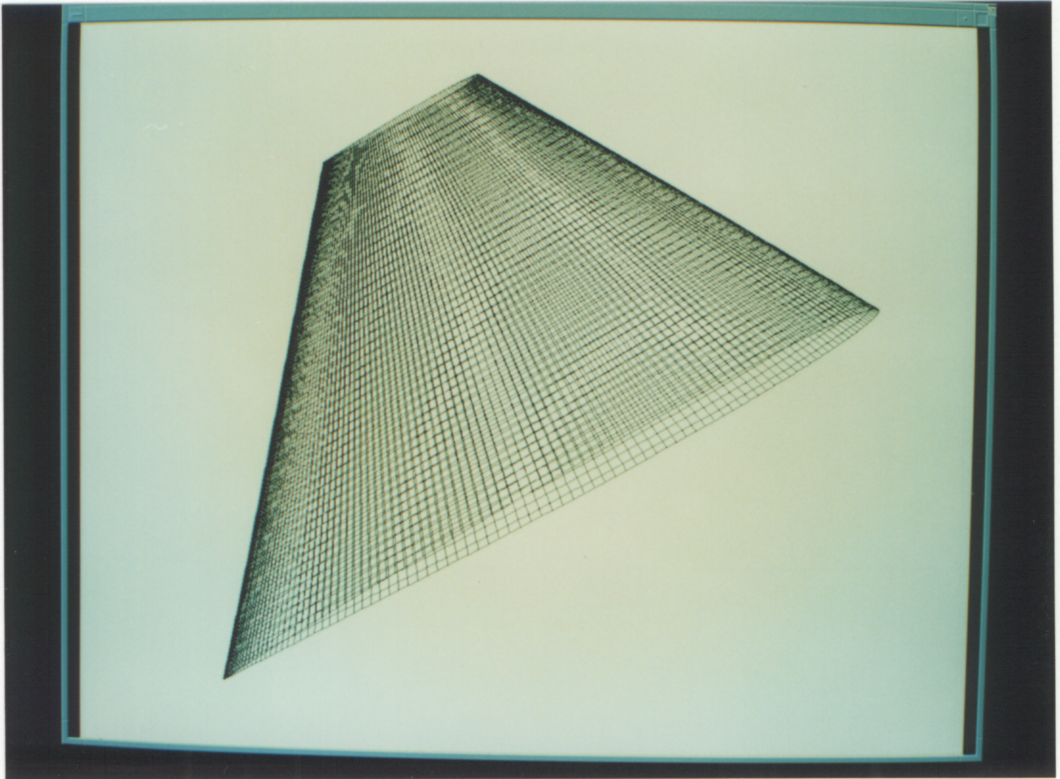


Figure 23. Original Wing Surface, 25 x 100 Vertex Control Polyhedron

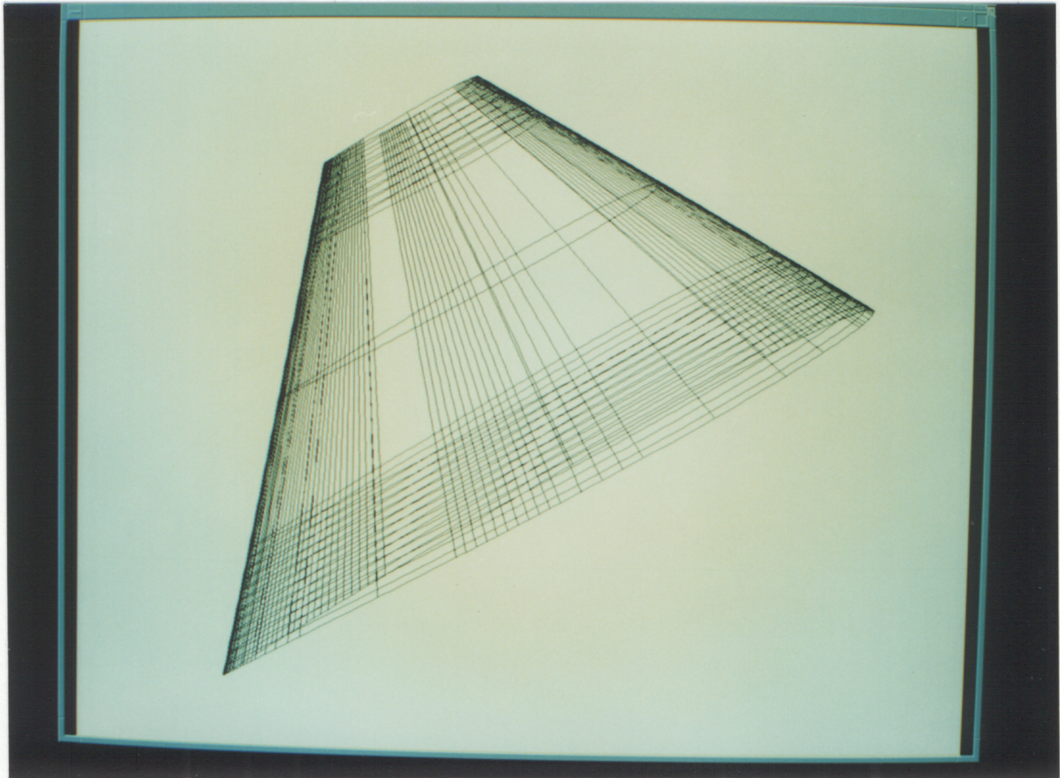


Figure 24. Reduced Wing Surface, 16 x 50 Vertex Control Polyhedron

8.0 Conclusions and Recommendations

8.1 Conclusions

The refinement and implementation of a data reduction and knot removal algorithm for non-uniform B-Spline curves and surfaces was presented. The results for curve-based data reduction show that the accuracy of the B-Spline representation was maintained to within approximately 10^{-3} inches (3×10^{-5} meters) on a curve measuring over 16 feet (4.877 meters) in length. However, no method for determining the accuracy of surface-based reduction was implemented. Even so, the knot removal algorithm which gave very good results for non-uniform B-Spline curves was used to perform knot removal for non-uniform B-Spline surfaces one cross section at a time. Accordingly, the results for surface-based data reduction are also expected to be accurate.

8.2 Recommendations

Although the data reduction algorithms for non-uniform B-Spline curves and surfaces presented in this thesis are robust, there is plenty of room for improvement. In particular, areas such as the technique used to classify knots for removal, the method for partitioning knots for removal, the user interface, and the calculation of error need to be improved.

A better technique for classifying knots for removal is needed. While the current method finds the locations of the closest knots, it does not consider curvature. Accordingly, knots which are chosen for removal may be located in an area of high curvature. In most cases, when these knots are removed, the local curvature is not retained. Therefore, a possible solution to this difficulty may be to include curvature as part of the classification process. Curvature could be incorporated into the classification process by using the product of the weight based on knot insertion and the curvature at the knot of interest as a new weight. Using the weight based on knot insertion and the curvature, k, the new weight could be defined as:

$$w_{j_{new}} = k \| f - g \|$$

Thus, knots along portions of the curve with low curvature would have lower weights, while knots along portions of the curve with high curvature would have higher weights. As an added improvement, it may be advantageous for the user to be able to specify which method is used to classify knots for removal. Three possible choices would be to classify knots for removal using weights based on:

- the knot insertion method for calculating weights only,
- curvature at knots along the curve only, and
- the product of the curvature and the knot insertion weight.

Another possible solution to the difficulties associated with classifying knots for removal could be to divide the classification process into two steps. In the first step, the knots defining the curve could be sorted based on curvature. Then the knots in areas having low curvature could be ranked for removal using weights based on knot insertion. These methods could also be applied to surfaces by classifying knots for removal on each isoparametric curve on the surface. While it would be convenient to be able to claim that one of these methods is best, the only way to be certain is to test the merits of each.

Another aspect of classifiying knots for removal which can be improved is the method used to find the knot removal weights for a row of knots on a surface. The method which is currently used to determine a weight for a row of knots is to average the weights of the knots forming the row. Perhaps a more representative knot removal weight can be found for a row of knots by using either the maximum or by using the least squares distribution of the weights for the knots in the row. This would lower the chances of a row of knots having both very high and very low weights from being removed.

In addition to developing an improved method of classifying knots for removal, a more adaptive partitioning technique is needed to determine the number of knots that should be removed during each iteration of the data reduction algorithm. No new methods have been devised to automatically determine the partition size. However, the partition size could be adjusted by the user. This would allow larger partition sizes to be specified

for quicker, less accurate data reductions, or smaller partition sizes to be specified for more exact, but slower data reductions. Yet another approach to partitioning would be to base the partition size on the number of knots in an area having low curvature. In any case, the creation of new methods for determining the partition size is necessary if the data reduction algorithm is to become faster. Larger partition sizes will yield a faster algorithm, while smaller partition sizes will result in a slower data reduction algorithm.

While being able to adjust the partition size will allow the user to control the speed of the data reduction algorithm, the user interface should support designers having a basic background as well as an extensive background on non-uniform B-Spline curves and surfaces. Default values have already been provided for users having only a basic knowledge of non-uniform B-Spline curves and surfaces. However, options such as being able to choose the number and which knots or rows of knots, will be removed should be available for designers who wish to make such choices. Another improvement would be to allow the user to specify the area of a component or curve which is to be reduced using a rubber band bounding box. This may entail subdividing the component upon which data reduction is being performed into two smaller components. Since these changes would significantly enhance the amount of control the user would have over the data reduction process, the opposite operation of refining a non-uniform B-Spline curve or surface should be available with the same options in the user interface. In essence, this would provide the designer with not only a technique for data reduction, but also with the tools necessary to fair non-uniform B-Spline curves and surfaces.

The final improvement which should be made to the data reduction algorithm is the calculation of the error for the reduced surface and the comparison of this error with a specified tolerance, ε. The user should be able to easily identify what tolerance is being

used and the error associated with the reduced curve or surface. Another feature which would help the designer visualize the effects of data reduction on the accuracy of the surface representation would be to map the error to the surface during display using a color bar and a numerical scale.

9.0 References

[Bart87] Bartels, R. H., Beatty, J. C. and Barsky, B. A., "An Introduction to Splines for Use in Computer Graphics and Geometric Modeling", *Morgan Kaufmann Publishers*, Los Altos, California, 1988.

[Boeh80] Boehm, W., Hartmut, P., "Inserting New Knots into B-Spline Curves", *Computer Aided Design*, 12, No. 4, 1980, pp.199-201.

[Boeh85] Boehm, W., Prautzsch, H., "The Insertion Algorithm", *Computer Aided Design*, 17, No. 2, 1985, pp.58-59.

[Cohe80] Cohen, E., Lyche, T., Riesenfeld, R., "Discrete B-Splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Grahpics", *Computer Graphics and Image Processing*, 14, 1980, pp.87-111.

[DeBo72] De Boor, C., "On Calculating with B-Splines", *Journal of Approximation Theory*, 6, 1972, pp.50-62.

[DeBo76] deBoor, C., "Splines as Linear Combinations of Splines", *A Survey, in Approximation Theory II* (Lorentz, G., Chui, C. K., and Schumaker, Eds.), *Academic Press*, New York, 1988.

[Fari87] Farin, G., Rein, G., Sapidis, N., Worsey, A. J., "Fairing Cubic B-Spline Curves", *Computer Aided Geometric Design*, 4, 1987, pp.91-103.

[Fari90a] Farin, G., Sapidis, N., "Automatic Fairing Algorithm for B-Spline Curves", *Computer Aided Design*, 22, No. 2, March 1990, pp.121-129.

[Fari90b] Farin, G., "Curves and Surfaces for Computer Aided Geometric Design - A Practical Guide", *Academic Press*, New York, 1990.

[Greg73] Gregory, T. J., "Computerized Preliminary Design at the early Stages of Vehicle Definition", NASA TM X-62,303, 1973.

[John82] Johnson, L. W. and Riess, R. D., "Numerical Analysis", *Addison-Wesley*, London, 1982.

[Kjel83] Kjellander, J. A., "Smoothing of Cubic Parametric Splines", *Computer Aided Design*, 15, No. 3, 1983, pp.175-179.

[Lych75] Lyche, T., "Discrete Polynomial Spline Approximation Methods", Thesis, University of Texas at Austin, 1975.

[Lych76] Lyche, T., "Discrete Cubic Spline Interpolation", *BIT*, 16, 1976, pp.281-290.

[Lych87a] Lyche, T. and Morken, K., "A Discrete Approach to Knot Removal and Degree Reduction Algorithms for Splines", *Algorithms for Approximation* (Mason, J. C., and Cox, M. G., Eds.), *Oxford University Press*, Oxford, 1987.

[Lych87b] Lyche, T., and Morken, K., "Knot Removal for Parametric B-Spline Curves and Surfaces", *Computer Aided Geometric Design*, 4, 1987, pp.217-230.

[Lych88] Lyche, T., and Morken, K., "A Data-Reduction Strategy for Splines with Application to the Approximation of Functions and Data", *IMA Journal of Numerical Analysis*, 8, 1988, pp.185-208.

[Mort85] Mortenson, M., "Geometric Modeling", *John Wiley and Sons*, 1985.

[Schu73] Schumaker, L., "Constructive Aspects of Discrete Polynomial Spline Functions" , *in Approximation Theory* (Lorentz, C. C.), *Academic Press*, New York, 1973.

[Wamp88a] Wampler, S., "Development of a CAD System for Automated Conceptual Design of Supersonic Aircraft", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Wamp88b] Wampler, S., Myklebust, A., Jayaram, and Gelhausen, P., "Improving Aircraft Conceptual Design - A PHIGS Interactive Graphics Interface For ACSYNT", *American Institute of Aeronautics and Astronautics*, Aircraft Designs, Systems and Operations Conference, AIAA-88-4481, 1988.

[Yama88] Yamaguchi, F., "Curves and Surfaces in Computer Aided Geometric Design", *Springer-Verlag*, 1988.

Appendix A. Norms

Vector norms are used extensively in the data reduction and knot removal algorithms presented in this thesis. Accordingly, a good understanding of norms is important. This appendix provides a short review of vector norms and describes the norms used in this thesis.

A.1 Vector Norms

When solving of a linear system of equations, the error of the solution is usually a concern. In most cases, it is hoped that the error is small. However, many occasions arise when the error in a solution is large. Since the terms large and small are relative, a quantitative way of determining when the error is large or small is needed. Vector norms are one method of quantitatively measuring the error in a solution to a system of linear equations. Vector norms can also be used to assess the size or the distance

between two vectors. Accordingly, norms serve as a method to extend the concepts of absolute value or magnitude from real numbers to vectors [John82].

A norm in $\mathbf{R}^n$ is a real-valued function satisfying three conditions:

- $||\mathbf{x}|| \geq 0$ and $||\mathbf{x}|| = \mathbf{0}$ if and only if $\mathbf{x} = \mathbf{0}$;
- $||\alpha\mathbf{x}|| = |\alpha| \, ||\mathbf{x}||$, for all scalars α and vectors $\mathbf{x}$;
- $||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}||$, for all vectors $\mathbf{x}$ and $\mathbf{y}$.

The three usual ℓ_p norms are defined as:

$$||\mathbf{x}||_1 = |x_1| + |x_2| + \ldots + |x_n|$$

$$||\mathbf{x}||_2 = \sqrt{(x_1)^2 + (x_2)^2 + \ldots + (x_n)^2}$$

$$||\mathbf{x}||_\infty = \max \{ |x_1|, |x_2|, \ldots, |x_n| \}$$

These are commonly referred to as the ℓ_1, ℓ_2, and the ℓ_∞ norms. The ℓ_∞ norm is also called the max norm since its value is the maximum component of the vector. For a thorough review of vector norms see [John82].

A.2 Norms Used for Knot Removal

The ℓ_p norms described in the previous section can be modified to suit the context in which they are being used. The norms which are used in this thesis are based on a modified form of the following norm:

$$||\mathbf{x}||_p = (\sum_j |x_j|^p)^{1/p}$$

which is similar to the ℓ_2 norm presented earlier. Since B-Splines are affected by both control vertices and a knot sequence, the norm used to calculate new control vertices for a B-Spline, f, during knot removal is:

$$||\mathbf{f}|| = \begin{cases} \sum_j |\mathbf{x}_j|^p [(t_{j+k} - t_j)/k]^{1/p} & for\ 1 \leq p < \infty, \\ \max_j |\mathbf{x}_j| & for\ p = \infty \end{cases}$$

where the $\mathbf{x}_j$ and t_j are the control vertices and knots, respectively, of the B-Spline curve. This norm gives a value representing the cumulative relationship of the control vertices, $\mathbf{x}_j$, to the change in surrounding knot values from knot t_j to knot t_{j+k} when $p = 2$. Refer to Figure 3 on page 24 for a better understanding of how the change in knot values relates to the control vertices. For a more extensive discussion of these norms see [Lych88].

Appendix B. Source Code Listing

B.1 Classification of Knots for Removal

```
/**********************************************************************
*    Name:  Boehm_rat.c
*    Author:  Fred W. Marcaly
*    Date:  October 4, 1990
*
*    Description:  Contains modules to compute the knot removal ratio
*                  derived in [Boeh80].
*
**********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations-------------------------*/

 float
      Boehm_ratio();

/*----------------------End of Function Declarations------------------*/

/*================================================================*
* Module Name:  Boehm_ratio()
*=================================================================*
* Description:  Calculates the knot insertion ratio (mu) derived by Boehm.
*
* Input:        Knot vector for the curve with the knot value already
*               removed, value of the knot being removed, number of the
*               control point for which the Boehm_ratio is to be
*               computed and the order of the B-Spline.
*
* Output:       Boehm_ratio.
*=================================================================*/
float Boehm_ratio(knot, value_removed, i, k)
 float
     knot[],    /* knot vector for the curve with the knot value already
                   removed                                            */
     value_removed;                 /* knot value which was removed */
 int
```

```
    i,          /* number of the control point for which the Boehm_ratio
                   is being calculated (1...n)                        */
    k;                                              /* order of B-spline */
{
 float
       mu,                               /* Boehm_ratio being calculated */
       denom;                        /* denominator of expression for mu */


 if ( (denom = knot[i+k-2] - knot[i-1]) > 0.01)
    mu = (value_removed - knot[i-1])/denom;

 else
    mu = -1.;     /* error condition */

 return (mu);
}

/***********************************************************************
*    Name:  avg_weights.c
*    Author:  Fred W. Marcaly
*    Date:  March 6, 1991
*
*    Description:  Contains modules to find average knot removal weights
*                  for a surface.
*
***********************************************************************/

#include<afmnc.h>
#include<stdio.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations-------------------------*/

void
  avg_w_comp_weights();

/*----------------------End of Function Declarations------------------*/

/*===================================================================*
* Module Name:  avg_w_comp_weights()
*====================================================================*
* Description:  Finds the average knot removal weights for knot removal
*               in the w parametric direction for a surface and stores
*               the average weights in cross section zero of the rem_data
*               structure for the component.
*
* Input:        Pointer to the component.
*
* Output:       None.
*===================================================================*/
void avg_w_comp_weights(Model)
 MODEL
     *Model;  /* pointer to the component whose weights are being averaged */
{
 int
   k = 4,                                  /* order of B-Spline curve */
   xsn,              /* cross section number whose weight is being found */
   j,       /* subscript of row of w knots whose average weight is being
               found                                                   */
   count;                      /* counts the number of weights in the sum */
 comp_data
   *comp,     /* pointer to the current component whose average weights are
               being found                                             */
   *prev_comp;                           /* pointer to the previous component */

 comp = Model->comp;
```

```
 while (comp != (comp_data *)NULL)
   {
    /* for each interior control point on a curve */
    for (j = k+1+4; j < comp->nw_knots-k-2-4; j++)
      {
       count = 0;

       /* for each cross section of the component */
       for (xsn = 0; xsn < comp->nu_knots - 4; xsn++)
         {
          /* sum knot removal weights */
          comp->rem[0][j]->weight += comp->rem[xsn][j]->weight;
          count++;
         }

       /* find average weight and store in cross section zero */
       comp->rem[0][j]->weight = comp->rem[0][j]->weight/(float)count;
      }

    /* go on to next component */
    prev_comp = comp;
    comp = prev_comp->next;
   }

}

/**********************************************************************
*    Name:  free_weight.c
*    Author:  Fred W. Marcaly
*    Date:  10/11/90
*
*    Description:  Contains module to free the Weight data structure.
*
**********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
#include<stdio.h>
#include<math.h>

/*-----------------------Function Declarations-------------------------*/

 void
     free_weight_struct();

/*----------------------End of Function Declarations------------------*/

/*===================================================================*
* Module Name:  free_weight_struct()
*===================================================================*
* Description:  Frees memory in the Weight data structure and mu.
*
* Input:        Pointer to the Weight data structure and mu.
*
* Output:       Freed Weight data structure.
*===================================================================*/
void free_weight_struct(Weight, mu)
 WEIGHT
        *Weight;                      /* pointer to Weights data structure */
 float
        *mu;             /* pointer to array of mu values used to find p */
{

 free(Weight->knot);
 free(Weight->hull);
 free(Weight->trans);
```

```
 free(Weight->rhsx);
 free(Weight->rhsy);
 free(Weight->rhsz);
 free(Weight->solnx);
 free(Weight->solny);
 free(Weight->solnz);
 free(mu);

}

/*************************************************************************
*    Name:  largest_p.c
*    Author:  Fred W. Marcaly
*    Date:  October 4, 1990
*
*    Description:  Contains modules to find the largest p such that
*                  MUp >= 1/2.  See [Lych87].
*
*************************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations-------------------------*/

 int
     largest_p_of_MUp();

/*----------------------End of Function Declarations------------------*/

/*====================================================================*
* Module Name:  largest_p_of_MUp()
*====================================================================*
* Description:  Finds the largest p such that MUp >= 1/2.
*
* Input:        Number of control points defining the curve after
*               removing a knot and array of MU values.
*
* Output:       Largest p such that MUp >= 1/2.
*====================================================================*/
int largest_p_of_MUp(n,MU)
 int
     n;                    /* number of control points defining the curve
                              after removing a knot                    */
 float
       MU[];                          /* MU values for each control point */

{

 int
     i,
     count = 0,
     xa, /* lower index for range of MU */
     xb, /* upper index for range of MU */
     a;  /* middle index for range of MU */


 /* initialize range for search on MU */
 xa = 0;
 xb = n-1;

 /* initialize center of range for search */
 a = (xb - xa)/2;

 /* loop until search converges on or very near MU = 1/2 */
 while ( (xb - xa)>1 )
   {
```

```
    count++;
    if (MU[a] > 0.5)
      xa = a;
    else if (MU[a] < 0.5)
      xb = a;
    else
      if (xa == a && MU[xb+1] == 0.5)
        {
        xa = xb = xa+1;
        }
      else if (xb == a && MU[xb-1] == 0.5)
        {
        xb = xa = xb-1;
        }
      else
        {
        xa = xb = a;
        }

    /* reset middle of range to search */
    a = xa + (xb - xa)/2;
   }
 return(xa+1);
}

/************************************************************************
*    Name:  main_weight.c
*    Author:  Fred W. Marcaly
*    Date:  10/11/90
*
*    Description:  Contains modules needed to calculate the weights
*                  used to determine which knots should be removed
*                  during data reduction.
*
************************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
#include<stdio.h>
#include<math.h>

/*----------------------Function Declarations-------------------------*/

 void
     message(),
     weight_alloc(),
     weight_init(),
     free_weight_struct(),
     fwd_Gauss_elim(),
     bwd_Gauss_elim(),
     solve(),
     weights(),
     transf_mat(),
     rem_alloc(),
     curve_weight();
 float
     max_norm(),
     set_weight();

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  weights();
*=================================================================*
* Description:  Main driver routine for calculating the weights used to
*               determine which knots should be removed.
```

```
*
* Input:        Pointer to the model containing the B-Spline curve from
*               which knots are to be removed.
*
* Output:       All weights are stored in comp->weight[][].
*=================================================================*/
void weights(Model,Weight)
MODEL
     *Model;                /* pointer to the model from which knots are
                               to be removed                          */
WEIGHT
     *Weight;               /* pointer to data structure used to find
                               knot removal weights                   */
{

 int
   err,                                            /* error indicator */
   v,                                   /* index of knot being removed */
   start,       /* starting subscript in MODEL hull data structure to
                begin copying control hull into WEIGHT data structure */
   size,                                    /* amount of memory needed */
   num_col,                      /* number of unknowns in the system
                                   of equations used to find weights */
   count,                             /* index for rhs and soln arrays */
   i,                      /* loop variable for number of components */
   compn=1,   /* component number from which knots are to be removed */
   xsn,           /* curve number from which knots are to be removed */
   uw=1,                          /* flag for curve type:
                                      uw = 0 -> u curve
                                      uw = 1 -> w curve               */
   L = 1,                              /* assumed knot multiplicity */
   k = 4,                                     /* degree of B-Spline */
   j,                                             /* loop variable */
   xyz;                                           /* loop variable */

 float
   *mu;                /* pointer to array of mu values used to find p */

 comp_data
   *comp,                   /* pointer to current component in Model */
   *prev_comp;             /* pointer to previous component in Model */


 /* traverse components in data structure */
 for (i = 0; i<Model->num_comp; i++)
   {
   if (i == 0)
       comp = Model->comp;
   else
       comp = prev_comp->next;

      /* correct component number ==> determine weights */

      /* set number of u and w knots being removed */
      comp->nu_removed = 0;
      comp->nw_removed = 1;

      /* allocate memory for knot removal weights for the current curve */
      rem_alloc(comp);

      /* set number of knots remaining if a single knot is removed */
      Weight->n_knots = comp->nw_knots-1;

      /* set number of unknowns in the system of equations */
      num_col = k-L+2;
```

```
        /* allocate memory for mu */
        mu = (float *)calloc(k-L+2,sizeof(float));

        /* allocate memory for knots, hull, transformation
           matrix, right hand side matrix and solution matrix    */
        weight_alloc(Weight, num_col);


        /* ---loop through each curve in component--- */
        if ( comp->nu_knots-4 > 0 )
          {
           /* component is a surface */
           for (xsn = 0; xsn < comp->nu_knots-4; xsn++)
             {
                /* determine knot removal weight for each interior knot
                   on curve                                             */
                curve_weight(Weight, comp, num_col, xsn, mu, k, L);

             } /* end of xsn loop */
          }
        else if ( comp->nu_knots-4 < 1 )
          {
           /* component is a curve in the w parametric direction */
           xsn = 0;
           curve_weight(Weight, comp, num_col, xsn, mu, k, L);
          }


          /* free Weights data structure */
          free_weight_struct(Weight,mu);

     /* change current component to previous component */
     prev_comp = comp;

    }
}
/*=================================================================*
* Module Name: curve_weight()
*=================================================================*
* Description: Determines the knot removal weights used during
*              data reduction.
*
* Input:       Pointer to the Weight data structure, pointer to the
*              component data structure, number of unknowns in the system
*              of equations being solved to find weights, number of the
*              curve whose weights are being found, array for mu values,
*              order of curve and knot multiplicity.
*
* Output:      Weight values assigned to the correct row of a 2-d weight
*              matrix for a component.
*=================================================================*/
void curve_weight(Weight, comp, num_col, xsn, mu, k, L)
 WEIGHT
       *Weight;              /* pointer to the Weight data structure */
 comp_data
         *comp;           /* pointer to the component data structure */
 int
    L,                                          /* knot multplicity */
    k,                                         /* order of B-Spline */
    num_col,        /* number of unknowns in system of equations being
                       solved to find weights                        */
    xsn;              /* number of curve whose weights are being found */
 float
      mu[];                                   /* array for mu values */


{
```

```
   for (Weight->rem_index = k+1+4; Weight->rem_index <= comp->nw_knots-k-2-4;
  Weight->rem_index++)

      {

       /* initialize index of knot to be removed                    */
       Weight->rem_val = comp->w_knot[Weight->rem_index-1];

       /* initialize knot vector, control hull, right hand side matrix
          and transformation matrix */
       weight_init(comp, Weight, num_col, xsn, mu);

       /* perform forward Gaussian elimination down to the pth row */
       fwd_Gauss_elim(Weight,num_col,num_col,Weight->trans,Weight->rhsx,
                     Weight->rhsy,Weight->rhsz,Weight->p);

       /* perform backward Gaussian elimination up to the p+1th row */
       bwd_Gauss_elim(Weight,num_col,num_col,Weight->trans,Weight->rhsx,
                     Weight->rhsy,Weight->rhsz,Weight->p);

       /* solve system of equations */
       solve(Weight, num_col);

       /* set weight */
       comp->rem[xsn][Weight->rem_index-1]->weight = set_weight(Weight);
       comp->rem[xsn][Weight->rem_index-1]->index = Weight->rem_index;

      } /* end of for Weight->rem_index loop */
}

/*********************************************************************
*    Name:  partition.c
*    Author:  Fred W. Marcaly
*    Date:  February 27, 1991
*
*    Description:  Contains modules which determine partition size and
*                  the knot removal list for a B-Spline curve.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
#include "../execs/approximate.h"
#include<stdio.h>

/*----------------------Function Declarations------------------------*/

 int
     find_partition_size(),
     make_knot_removal_list();

/*----------------------End of Function Declarations-----------------*/

/*==================================================================*
* Module Name:  find_partition_size()
*===================================================================*
* Description:  Determines the next partition size for removing the
*               requested number of knots from the curve.
*
* Input:        Pointer to the Model containing the curve and total number
*               of knots to be removed from the curve.
*
* Output:       Partition_size.
*===================================================================*/
 int find_partition_size(Model, total_num_removed, num_to_be_removed)
 MODEL
```

```
     *Model;                                   /* pointer to the model */
int
     num_to_be_removed,          /* total number of knots to be removed */
     total_num_removed;               /* number of knots remove so far */
{
int
     partition_size;

/* make partition size <= 5 */
partition_size = num_to_be_removed - total_num_removed;

if (partition_size > 5)
   partition_size = 5;

return (partition_size);

}
/*==================================================================*
* Module Name:  make_knot_removal_list ()
*==================================================================*
* Description:  Makes all knot removal lists for a partition.
*
* Input:        Pointer to the array of rem_data structures for the
*               cross section being reduced, partition size, total
*               number of knots removed so far, pointer to an array of
*               knot removal lists.
*
*
* Output:       Returns the number of knot removal lists created.
*==================================================================*/
int make_knot_removal_list(rem, partition_size, total_num_removed,
                           removal_list,num_in_removal_list)
rem_data
    **rem;  /* pointer to a one dimensional array of pointers to rem_data
               structures                                             */
int
    partition_size,              /* length of partition being removed */
    total_num_removed,         /* total number of knots removed so far */
    **removal_list,                     /* 5x5 array of removal lists */
    num_in_removal_list[];     /* number of knots in each removal list */
{
int
   i, j,
   num_skipped = 0,      /* number of indices skipped between starting
                            new removal lists                          */
   num_lists = 0,           /* number of knot removal lists created */
   start_index,         /* rem array index of first knot in partition */
   end_index,          /* rem array index of last knot in partition    */
   num_added = 0,      /* number of knots added to a knot removal list */
   num_removed_from_partition = 0; /* number of knots removed from the
                                      partition                        */

/* start a removal list for each knot in the partition, assuming
   each knot may be put in a unique removal list */

start_index = 12;  /* adjust for zero weights */
end_index = start_index + partition_size;

for (i = start_index; i < end_index; i++)
  {
   num_added = 0;    /* initialize number of knots added to removal list */

   if (rem[i]->index != -1)
     {
      /* --- knot has not been put in a removal list yet --- */
      /* increment number of removal lists */
      num_lists++;
```

```
        /* begin removal list */
        removal_list[i-start_index-num_skipped][0] = rem[i]->index;

        for (j = i+1; j < end_index; j++)
          {
           if(rem[j]->index >=
              removal_list[i-start_index][0] - partition_size + 1 &&
              rem[j]->index <=
              removal_list[i-start_index][0] + partition_size - 1 &&
              rem[j]->index != -1)
             {
              num_added++;
              removal_list[i-start_index][num_added] =
                   rem[j]->index;
              rem[j]->index = -1;
             }
          } /* end for j loop */
       num_in_removal_list[i-start_index] = num_added + 1;
       } /* end if */
     else
       {
        num_in_removal_list[i-start_index] = 0;
        num_skipped++;
       }

       num_in_removal_list[i-start_index] = num_added + 1;
    } /* end for i loop */

  /* adjust removal lists to account for knots being removed in previous
     removal lists */
  for(i = 1; i < num_lists; i++)
    {
     num_removed_from_partition += num_in_removal_list[i-1];
     for (j = 0; j < num_in_removal_list[i]; j++)
        removal_list[i][j] -= num_removed_from_partition;
    }

  return(num_lists);
}

/***********************************************************************
*    Name:  rank.c
*    Author:  Fred W. Marcaly
*    Date:   January 30, 1991
*
*    Description:  Contains modules to sort the knots of a B-Spline curve
*                  based on their knot removal weights during knot
*                  removal.
*
*    Note:  The basic function of these modules is to partition and sort
*           the list of weights and control point indices which is
*           produced in the weight() module.  The initial partition for
*           sorting the list of weights is the maximum allowable weight
*           for which a knot can be removed.  The initial list of weights
*           is partitioned into two smaller lists using the maximum
*           allowable value as a dividing point.  All values less than or
*           equal to the maximum allowable value are placed in a list
*           which is then sorted using a quick sort.  The result is a
*           list of structures containing weights and corresponding
*           indices in order from lowest to highest weight, but less than
*           or equal to the maximum allowable weight.
*
***********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
```

```
/*----------------------Function Declarations-------------------------*/

 void
      rank( ),
      rank_component( ),
      rank_quicksort( ),
      rank_swap( );
 int
      rank_partition( ),
      rank_find_pivot( );

/*----------------------End of Function Declarations------------------*/

/*===================================================================*
* Module Name:  rank( )
*====================================================================*
* Description:  Ranks the control points of each component in a Model
*               for knot removal.  The array of rem_data structures for
*               each component is sorted so that the control points
*               that are the best candidates for knot removal are at
*               the beginning of the array comp->rem.
*
* Input:        Pointer to the Model to be ranked.
*
* Output:       None.
*====================================================================*/
void rank(Model)
 MODEL
      *Model;                     /* pointer to the model to be ranked */
{
 int
    i;                                             /* loop variable */
 float
    max_allowable_weight = 0.99; /* maximim allowable weight for a
                                   control point to be removed        */
                                 /* Note: This has been included for
                                          the implementation of adaptive
                                          partitioning.              */
 comp_data
    *comp,               /* pointer to component currently being ranked */
    *prev_comp;            /* pointer to last component that was ranked */

 for ( i = 0; i < Model->num_comp; i++)
   {
    if ( i ==0 )
      comp = Model->comp;
    else
      comp = prev_comp->next;

    rank_component(comp,4,comp->nw_knots-6-1,max_allowable_weight);

    prev_comp = comp;
   }
}
/*===================================================================*
* Module Name:  rank_component( )
*====================================================================*
* Description:  Controls initial partitioning and subsequent sorting of
*               the list of weights and control point indices of a single
*               component for knot removal.
*
* Input:        Pointer to the component whose weights are being sorted.
*
* Output:       None.
*====================================================================*/
void rank_component(comp, i, j, max_allowable_weight)
```

```
comp_data
          *comp;    /* pointer to the component whose weights are being
                       ranked                                          */
int
   i,    /* array index of first rem_data structure to be considered in
            ranking                                                    */
   j;    /* array index of last rem_data structure to be considered in
            ranking                                                    */
float
   max_allowable_weight; /* maximum allowable weight for a knot to
                            be removed                                 */
{
int
   xsn,                        /* cross section number on a component */
   k;           /* subscript of the pivot element after partitioning */
rem_data
        **rem;      /* 1-d array of pointers to structures
                       containing weight and index data               */


/* rank each curve in the w parametric direction for the component */
if ( comp->nu_knots-4 > 0)
  {
   for (xsn = 0; xsn < comp->nu_knots-4;  xsn++)
      {
       /* partition initial list of weights using the maximum allowable
          weight value */
       k = rank_partition(i,j,max_allowable_weight,&(comp->rem[xsn][0]));

       /* quick sort lower half of partitioned list of weights */
       rank_quicksort(i,k-1,comp->rem[xsn]);
      }
  }
else if (comp->nu_knots-4 == 0)
  {
   xsn = 0;
   /* partition initial list of weights using the maximum allowable
      weight value */
   k = rank_partition(i,j,max_allowable_weight,&(comp->rem[xsn][0]));

   /* quick sort lower half of partitioned list of weights */
   rank_quicksort(i,k-1,comp->rem[xsn]);
  }

}
/*===================================================================*
* Module Name:  rank_quicksort()
*====================================================================*
* Description:  Performs a quick sort on the array of rem_data structures
*               passed in between subscripts i and j.
*
* Input:        Indices of the first and last elements of the list to be
*               sorted.
*
* Output:       Sorted list.
*===================================================================*/
void rank_quicksort(i, j, A)
int
    i,           /* index of the first item in the list to be sorted */
    j;            /* index of the last item in the list to be sorted */
rem_data
    **A;         /* 1-d array containing a list of pointers to the
                    structures to be sorted                           */
{

float
    pivot;                              /* value of the pivot item */
```

```
 int
     debug,
     pivotindex,                          /* index of the pivot item */
     k;              /* beginning index for group of elements >= pivot */

 pivotindex = rank_find_pivot(i,j,A);

 if (pivotindex != -1)
   {
    pivot = A[pivotindex]->weight;
    k = rank_partition(i,j,pivot,A);
    rank_quicksort(i,k-1,A);
    rank_quicksort(k,j,A);
   }

}

/*======================================================================*
* Module Name:  rank_find_pivot()
*======================================================================*
* Description:  Returns the pivot value for the next partitioning
*               iteration of the quick sort.
*
* Input:        Indices of the first and last elements of the list to be
*               partitioned during the current iteration and the list to
*               be partitioned.
*
* Output:       Index of the item which is to be the pivot.
*======================================================================*/
 int rank_find_pivot(i,j,A)
 int
     i,   /* index of the first structure in the list to be partitioned */
     j;    /* index of the last structure in the list to be partitioned */
 rem_data
     **A;          /* 1-d array containing a list of pointers to the
                        structures to be sorted                         */
{
 int
     k;                   /* runs right looking for another item value */
 float
     firstKey;                  /* value of the first item in the list */

 firstKey = A[i]->weight;

 for(k = i+1; k <= j; k++)
   {
    if (A[k]->weight > firstKey)
      {
       return(k);
      }
    else if (A[k]->weight < firstKey)
      {
       return(i);
      }
   }

 return(-1);

}

/*======================================================================*
* Module Name:  rank_partition()
*======================================================================*
* Description:  Partitions the list around the pivot so that all numbers
*               greater than the pivot are on the right of the pivot and
*               all numbers less than the pivot are on the left of the
*               pivot.
```

```
*
* Input:        Indices of the first and last elements of the list to be
*               partitioned, pivot and the list to be partitioned.
*
* Output:       Partitioned list.
*=================================================================*/
 int rank_partition(i,j,pivot,A)
 int
     i,        /* index of the first item in the list to be partitioned */
     j;         /* index of the last item in the list to be partitioned */
 float
     pivot; /* value of the item about which partitioning is to be done */
 rem_data
     **A;           /* 1-d array containing a list of pointers to the
                        structures to be sorted                        */

{

 int
     L,
     R;
 rem_data
     *temp;

 L = i;
 R = j;

 while (R >= L && R>i && L<j)
   {
      /* swap values */
      temp = A[L];
      A[L] = A[R];
      A[R] = temp;

    while (A[L]->weight < pivot && L <=j)
      L++;
    while (A[R]->weight >= pivot && R >=i)
      R--;
   }

 return(L);
}

/******************************************************************
*    Name:  rem_alloc.c
*    Author:  Fred W. Marcaly
*    Date:  January 30, 1991
*
*    Description:  Contains modules to allocate rem_data structure of a
*                  component.
*
******************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations------------------------*/
void
     rem_alloc();

/*----------------------End of Function Declarations-------------------*/

/*=================================================================*
* Module Name:  rem_alloc()
*=================================================================*
* Description:  Allocates a two dimensional array which contains pointers
*               to rem_data structures.  These data structures contain
```

```
*               the weight and ranking data for data reduction.
*
* Input:        Pointer to the component data structure and a one
*               dimensional array of pointers to rem_data structures.
*
* Output:       None.
*==================================================================*/
void rem_alloc(comp, rem)
 comp_data
          *comp;                /* pointer to component data structure */
 rem_data
         **rem;         /* 1-d array of pointers to rem_data structures */
{
 int
   i,j;                                         /* loop variables */

  /*--- allocate component rem_data memory ---*/

  /* allocate pointers to rows */
  if ( comp->nu_knots - 4 > 0)
    {
     /* B-Spline surface */
     comp->rem = (rem_data ***)calloc(comp->nu_knots-4,sizeof(rem_data **));

     /* allocate pointers to elements of each row */
     for ( i = 0; i < comp->nu_knots-4; i++)
       {
        comp->rem[i] = (rem_data **)calloc(comp->nw_knots-4,sizeof(rem_data *));
        /* allocate space for pointer to the rem_data structure */
        for ( j = 0; j < comp->nw_knots-4; j++)
           comp->rem[i][j] = (rem_data *)calloc(1,sizeof(rem_data));
       }
    }
  else if ( comp->nu_knots - 4 <= 0)
    {
     /* B-Spline curve in w parametric direction */
     comp->rem = (rem_data ***)calloc(1,sizeof(rem_data **));

     /* allocate pointers to elements of each row */
     comp->rem[0] = (rem_data **)calloc(comp->nw_knots-4,sizeof(rem_data *));

     /* allocate space for pointer to the rem_data structure */
     for ( j = 0; j < comp->nw_knots-4; j++)
        comp->rem[0][j] = (rem_data *)calloc(1,sizeof(rem_data));
    }

  /*--- allocate memory for 1-d array of pointers to rem_data structures
        for ranking ---*/
  rem = (rem_data **)calloc(comp->nw_knots-4,sizeof(rem_data *));
  for ( i = 0; i < comp->nw_knots-4; i++)
     rem[i] = (rem_data *)calloc(1,sizeof(rem_data));
}
/*******************************************************************
*    Name:  solve.c
*    Author:  Fred W. Marcaly
*    Date:  10/10/90
*
*    Description:  Contains modules for back solving the system of n x n
*                  equations for determining knot removal weights used
*                  in data reduction.
*
*******************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
```

```
/*-----------------------Function Declarations--------------------------*/

 void
      solve( ),
      solve_xp_xn( ),
      fwd_solve( ),
      bwd_solve( );

/*-----------------------End of Function Declarations-------------------*/

/*=====================================================================*
* Module Name:  solve( )
*=====================================================================*
* Description:  Solves the set of equations to get the control point
*               solution needed for calculating knot removal weights.
* Input:        Pointer to the Weight data structure and number of
*               unknowns in system of equations
*
* Output:       Solved system of equations.
*=====================================================================*/
void solve(Weight, num_col)
 WEIGHT
       *Weight;                    /* pointer to weights data structure */
 int
     num_col;              /* number of unknowns in system of equations */
{
 int
     i,j;

 /* solve for Xp and Xn for x, y and z coordinates */
 solve_xp_xn(num_col,num_col,Weight->trans,Weight->rhsx,Weight->p,
            &Weight->solnx[Weight->p-1], &Weight->solnx[num_col-1]);

 solve_xp_xn(num_col,num_col,Weight->trans,Weight->rhsy,Weight->p,
            &Weight->solny[Weight->p-1], &Weight->solny[num_col-1]);

 solve_xp_xn(num_col,num_col,Weight->trans,Weight->rhsz,Weight->p,
            &Weight->solnz[Weight->p-1], &Weight->solnz[num_col-1]);


 /* back solve for X1 to Xp-1 for x, y and z coordinates */
 bwd_solve(num_col,num_col,Weight->trans,Weight->rhsx,
          Weight->p,Weight->solnx);

 bwd_solve(num_col,num_col,Weight->trans,Weight->rhsy,
          Weight->p,Weight->solny);

 bwd_solve(num_col,num_col,Weight->trans,Weight->rhsz,
          Weight->p,Weight->solnz);

 /* forward solve for Xp+1 to Xnum_col-1 for x, y and z coordinates */
 fwd_solve(num_col,num_col,Weight->trans,Weight->rhsx,
          Weight->p,Weight->solnx);

 fwd_solve(num_col,num_col,Weight->trans,Weight->rhsy,
          Weight->p,Weight->solny);

 fwd_solve(num_col,num_col,Weight->trans,Weight->rhsz,
          Weight->p,Weight->solnz);

}
/*=====================================================================*
* Module Name:  solve_xp_xn( )
*=====================================================================*
* Description:  Solves a system of two equations in two unknowns for
*               Xp and Xn.
*
```

```
* Input:         Number of unknowns in original system, number of
*                equations in original system, coefficient matrix on which
*                forward and the backward Gaussian elimination have been
*                performed up to the Pth unknown, right hand side matrix,
*                subscript of original system which Xp is to be solved for.
*
* Output:        Xp and Xn.
*=====================================================================*/
void solve_xp_xn(n,m,coeff,rhs,p,xp,xn)
int
   n,                          /* number of unknowns in original system */
   m,                         /* number of equations in original system */
   p;                               /* subscript of unknown to be found */

float
   coeff[], /* coefficient matrix on which forward and backward Gaussian
               elimination have been performed                          */
   rhs[],                           /* right hand side matrix of system */
   *xp,
   *xn;
{
 float
     mult;                    /* multiplier to eliminate xp from system */

 /* eliminate Xp from equations p and p+1 */
 mult = coeff[p*n+p-1] / coeff[(p-1)*n+p-1];

 coeff[p*n+n-1] = coeff[p*n+n-1] - mult*coeff[(p-1)*n+n-1];
 rhs[p] = rhs[p] - mult*rhs[p-1];

 /* solve second equation */
 *xn = rhs[p] / coeff[p*n+n-1];

 /* back substitute into first equation */
 *xp = (rhs[p-1] - coeff[(p-1)*n+n-1]*(*xn)) / coeff[(p-1)*n+p-1];
}
/*=====================================================================*
* Module Name:   bwd_solve()
*=====================================================================*
* Description:   Performs backwards substitution from the (P-1)th equation
*                to the 1st equation in a system of equations where the
*                Pth and (P+1)th unknowns have been found.
*
* Input:         Number of unknowns in the system, number of equations in
*                the system, coefficient matrix, right hand side matrix
*                and index where forward Gaussian elimination was stopped.
*
* Output:        Solution matrix.
*=====================================================================*/
void bwd_solve(n,m,coeff,rhs,p,x)
 int
    n,                                            /* number of unknowns */
    m,                                           /* number of equations */
    p;               /* index where forward Gaussian elim. was stopped */

 float
    coeff[],                                     /* coefficient matrix  */
    rhs[],                                    /* right hand side matrix */
    x[];                                             /* solution matrix */
{
 int
    i;                                 /* index of unknown being found */

 for (i = p-2; i >= 0; i--)
    x[i] = (rhs[i] - coeff[i*n+n-1]*x[n-1]) / coeff[i*n+i];
```

```
}
/*==================================================================*
* Module Name:  fwd_solve()
*==================================================================*
* Description:  Performs forwards substitution from the (P+1)th equation
*               to the Nth equation in a system of equations where the
*               Pth and (P+1)th unknowns have been found.
*
* Input:        Number of unknowns, number of equations, coefficient
*               matrix, right hand side matrix and the index where
*               Gaussain elimination was stopped.
*
* Output:       Solution matrix.
*==================================================================*/
void fwd_solve(n,m,coeff,rhs,p,x)
 int
    n,                                          /* number of unknowns */
    m,                                         /* number of equations */
    p;                  /* index where Gaussian elimination was stopped */

 float
    coeff[],                                   /* coefficient matrix */
    rhs[],                                  /* right hand side matrix */
    x[];                                          /* solution matrix */
{
 int
     i;                               /* index of unknown being found */

 for (i = p; i<n-1; i++)
    x[i] = (rhs[i+1] - coeff[(i+1)*n+n-1]*x[n-1]) / coeff[(i+1)*n+i];

}

/*******************************************************************
*    Name:  tool_weight.c
*    Author:  Fred W. Marcaly
*    Date:  10/11/90
*
*    Description:  Contains modules supporting the determination of
*                  knot removal weights.
*
*******************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
#include<stdio.h>
#include<math.h>

/*-----------------------Function Declarations-----------------------*/

 void
     weight_alloc(),
     weight_init(),
     transf_mat(),
     matrix_prod(),
     matrix_sub();
 int
     largest_p_of_MUp();
 float
     max_norm(),
     set_weight();

/*-----------------------End of Function Declarations-----------------*/

/*==================================================================*
* Module Name:  weight_alloc()
```

```
*======================================================================*
* Description:  Allocates memory needed for the determination of a weight
*               for each interior control point.
*
* Input:        Pointer to the Weight data structure, pointer for the
*               mu array used to find p and the number of columns in the
*               system of equations to be solved.
*
* Output:       Pointers to the allocated memory blocks.
*======================================================================*/
 void weight_alloc(Weight,num_col)
 WEIGHT
        *Weight;                   /* pointer to the Weight data structure */
 int
     num_col;          /* number of unknowns in the system of equations */
{
 int
     size;                        /* size of memory block to be allocated */


  /* allocate memory for knots, hull, MU, transformation
     matrix, right hand side matrix and solution matrix    */
  Weight->knot = (float *)calloc(Weight->n_knots, sizeof(float));
  size = 3*(Weight->n_knots-4);
  Weight->hull = (float *)calloc(size,sizeof(float));

  size = num_col*num_col;
  Weight->trans = (float *)calloc(size, sizeof(float));
  Weight->init_trans = (float *)calloc(size, sizeof(float));

  Weight->rhsx = (float *)calloc(num_col, sizeof(float));
  Weight->rhsy = (float *)calloc(num_col, sizeof(float));
  Weight->rhsz = (float *)calloc(num_col, sizeof(float));

  Weight->init_rhsx = (float *)calloc(num_col, sizeof(float));
  Weight->init_rhsy = (float *)calloc(num_col, sizeof(float));
  Weight->init_rhsz = (float *)calloc(num_col, sizeof(float));

  Weight->solnx= (float *)calloc(num_col, sizeof(float));
  Weight->solny= (float *)calloc(num_col, sizeof(float));
  Weight->solnz= (float *)calloc(num_col, sizeof(float));
}

/*=====================================================================*
* Module Name:  weight_init()
*======================================================================*
* Description:  Sets knot sequence, control hull, transformation matrix
*               and right hand side matrix needed to solve system of
*               equations for determining knot removal weights.
*
* Input:        Pointer to the component, pointer to the Weights data
*               structure, number of unknowns in the system of
*               equations used to find weights and cross section number
*               of component for which weights are being found.
*
* Output:       Initialized arrays Weight->knot, Weight->hull, and
*               Weight->rhsxyz.
*======================================================================*/
 void weight_init(comp, Weight, num_col, xsn, mu)
 comp_data
          *comp;                         /* pointer to the component data */

 WEIGHT
        *Weight;                  /* pointer to the Weight data structure */
 int
     num_col,            /* number of unknowns in the system of equations
                            being solved to find knot removal weights   */
```

```
    xsn;                 /* cross section number of component for which
                            weights are being found                       */
 float
      mu[];                          /* mu array used to find largest p */
{
 int
    i,                                                  /* loop variable */
    j,                                                  /* loop variable */
    xyz,                                                /* loop variable */
    start,        /* starting subscript in MODEL hull data structure to
                 begin copying control hull into WEIGHT data structure */
    v,               /* index of knot for which weight is being found */
    k = 4,                                         /* order of B-Spline */
    L = 1,                                     /* multiplicity of knots */
    count;                                    /* control point number */


  /* initialize knot sequence for finding weights */
  for (j = 0; j < Weight->n_knots; j++)
    {
     if (j+1 < Weight->rem_index)
       {
        Weight->knot[j] = comp->w_knot[j];
       }
  else
       {
        Weight->knot[j] = comp->w_knot[j+1];
       }
    }

  /* initialize control hull for finding weights */
  start = xsn*(comp->nw_knots-4)*3;
  for (j = 0; j < Weight->n_knots-4; j++)
     {
     for (xyz = 0; xyz < 3; xyz++)
        {
         if (j+1 < Weight->rem_index)
           {
            Weight->hull[3*j+xyz] = comp->hull[xsn][j][xyz];
           }
         else
           {
            Weight->hull[3*j+xyz] = comp->hull[xsn][j+1][xyz];
           }
        }
     }

  /* construct transformation matrix and mu array */
  transf_mat(Weight, mu);


  /* find the largest p such that mu >= 1/2 */
  Weight->p = largest_p_of_MUp(num_col, mu);

  /* initialize right hand side for finding weights */
  count = 0;
  v = Weight->rem_index;
  for (j = v-k-1; j < v-L+1; j++)
     {
      Weight->rhsx[count] = Weight->init_rhsx[count] = Weight->hull[3*j];
      Weight->rhsy[count] = Weight->init_rhsy[count] = Weight->hull[3*j+1];
      Weight->rhsz[count] = Weight->init_rhsz[count] = Weight->hull[3*j+2];
      count++;
     }
}
```

```
/*===================================================================*
* Module Name:  transf_mat()
*===================================================================*
* Description:  Sets up a matrix needed to transform the n control points
*               of a B-Spline to the n+1 control points of a different
*               B-Spline which represents the same curve data.
*
* Input:        Pointer to the Weight data structure.
*
* Output:       Transformation matrix (trans) set in Weight data
*               structure, and mu array of knot insertion ratios.
*===================================================================*/
void transf_mat(Weight, mu)
 WEIGHT
     *Weight;                          /* weight data structure pointer */
 float
     mu[];
{
 int
     j,
     subs,                                      /* subscript for mu */
     v,                               /* index of knot to be removed */
     k = 4,                                  /* order of B-Splines */
     L = 1,                               /* multiplicity of knots */
     i,                                   /* row and column index */
     ncol;                /* number of columns in transformation matrix */


 v = Weight->rem_index;
 ncol = (int)fabs((double)k-L+2);
 subs = v - k + 1;

 for (i = 0; i < ncol; i++)
   {
      if (i == 0)
        {
         /* first row */
         /* set mu */
         Weight->trans[i] = mu[i] = 1.0;

         /* set far right column */
         Weight->trans[ncol-1] = (float)pow((double)-1.0,(double)(ncol-1));

         for (j = 1; j < ncol-1; j++)
            Weight->trans[j] = 0.;

        }
      else if (i == (ncol - 1) )
        {
         Weight->trans[ncol*i+ncol-2] = Weight->trans[ncol*i+ncol-1] = 1.0;
         mu[i] = 0.;

         for (j = 0; j < ncol-2; j++)
            Weight->trans[ncol*i+j] = 0.;

        }
      else
        {
         for (j = 0; j < ncol-1; j++)
            Weight->trans[ncol*i+j] = 0.;

         /* set mu for current row */
         Weight->trans[ncol*i+i] = mu[i] =
                            Boehm_ratio(Weight->knot,Weight->rem_val,
                                            subs+i-1,k);
         /* set lamda for current row */
         Weight->trans[ncol*i+i-1] = 1.0 - Weight->trans[ncol*i+i];
```

```
          /* set far right column */
          Weight->trans[ncol*i+ncol-1] =
                          (float)pow((double)-1.0,(double)(ncol-i-1));
        }
  }

 /* store initial transformation matrix */
 for(i = 0; i < ncol*ncol; i++)
    Weight->init_trans[i] = Weight->trans[i];

}

/*=====================================================================*
* Module Name:   set_weight()
*=====================================================================*
* Description:   Determines the weight for a knot value.
*
* Input:         Pointer to the Weight data structure.
*
* Output:        Knot removal weight.
*=====================================================================*/
 float set_weight(Weight)
 WEIGHT
      *Weight;                 /* pointer to the Weight data structure */
{
 int
      err = 0,                         /* error flag from matrix_prod() */
      num_col,        /* number of unknowns in the system of equations */
      k = 4,                                       /* order of B-Spline */
      L = 1,   /* multiplicity of knot for which weight is being found */
      j,i;                                                 /* row index */

 float
      w_value,                  /* knot removal weight to be returned */
      *prodx,                   /* product of transformation and soln
                                   matrices in x coordinate           */
      *prody,                   /* product of transformation and soln
                                   matrices in y coordinate           */
      *prodz,                   /* product of transformation and soln
                                   matrices in z coordinate           */
      *diffx,                         /* difference of rhs and prod
                                         matrices in x coordinate     */
      *diffy,                         /* difference of rhs and prod
                                         matrices in y coordinate     */
      *diffz,                         /* difference of rhs and prod
                                         matrices in z coordinate     */
      *norm_mat;                        /* matrix used to find norm */


 /* set number of unknowns in system of equations used to find weight */
 num_col = k-L+2;

 /* allocate memory for product, difference and norm arrays */
 prodx = (float *)calloc(num_col,sizeof(float));
 prody = (float *)calloc(num_col,sizeof(float));
 prodz = (float *)calloc(num_col,sizeof(float));

 diffx = (float *)calloc(num_col,sizeof(float));
 diffy = (float *)calloc(num_col,sizeof(float));
 diffz = (float *)calloc(num_col,sizeof(float));

 norm_mat = (float *)calloc(num_col*3,sizeof(float));


 /* calculate product of transformation matrix and solution matrix */
 matrix_prod(num_col, num_col, num_col, 1,
```

```
            Weight->init_trans, Weight->solnx, prodx, err);
matrix_prod(num_col, num_col, num_col, 1,
            Weight->init_trans, Weight->solny, prody, err);
matrix_prod(num_col, num_col, num_col, 1,
            Weight->init_trans, Weight->solnz, prodz, err);

if ( err == 1)
  {
   printf("set_weight( )\tincorrect argument in call to matrix_prod\n");
   printf("set_weight( )\tnum_col = %d\n",num_col);
  }


/* calculate difference between original control vertices and new
   control vertices */

matrix_sub(num_col,1,Weight->init_rhsx,prodx,diffx);
matrix_sub(num_col,1,Weight->init_rhsy,prody,diffy);
matrix_sub(num_col,1,Weight->init_rhsz,prodz,diffz);


/* construct norm matrix */
for (i = 0; i < num_col; i++)
  {
   norm_mat[i] = diffx[i];
  }
for (i = num_col; i < 2*num_col; i++)
  {
   norm_mat[i] = diffy[i-num_col];
  }
for (i = 2*num_col; i < 3*num_col; i++)
  {
   norm_mat[i] = diffz[i-2*num_col];
  }

/* find weight by calculating the max norm */
w_value =  max_norm(Weight, 3*num_col, norm_mat);

/* free memory used to find weight_value */
free(diffx);
free(diffy);
free(diffz);
free(prodx);
free(prody);
free(prodz);
free(norm_mat);

return (w_value );
}
```

B.2 Calculation of New Control Vertices

```
/************************************************************************
*    Name:  approx_init.c
*    Author:  Fred W. Marcaly
*    Date:  December 11, 1990
*
*    Description:  Contains modules to initialize the data structure
*                  used in the calculation of new control vertices
*                  during knot removal.
*
************************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*-----------------------Function Declarations-------------------------*/

void
    approx_init( ),
    init_parameters_in_approx( ),
    init_hi_low_approx_indices( ),
    init_vertices( ),
    init_knots( ),
    init_insertion_matrix( );

/*----------------------End of Function Declarations-------------------*/

/*=====================================================================*
* Module Name:  approx_init( )
*=====================================================================*
* Description:  Main driver for initializing Approx data structure for
*               the calculation of new control vertices during knot
*               removal.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       Approx data structure initialized with original control
*               vertices and knot sequence.
*=====================================================================*/
void approx_init(comp, Approx, xsn)
comp_data
         *comp;                          /* pointer to component data */
APPROXIMATION
             *Approx;           /* pointer to the Approx data structure */
int
   xsn;     /* cross section number of curve from which knots are being
               removed                                               */
{
 int
    err = 0,           /* error flag from matrix multiplication routine */
    i, j, xyz;                                     /* loop variables */
 float
      **vertex_product,   /* array for temporary storage of x, y and
                             z components of product matrix          */
      **vertex_component; /* array for temporary storage of x, y and
                             z components of original control vertices   */


 /* initialize control hull */
 init_vertices(comp, Approx, xsn);

 /* initialize knot sequence */
 init_knots(comp, Approx);
```

```
/* construct scaling matrix for original knot sequence, t */
construct_scaling_matrix(Approx->k,Approx->m,2.,Approx->t,
                         Approx->scale_t);

/* initialize scaling matrix for reduced knot vector, tau */
construct_scaling_matrix(Approx->k,Approx->n,-2.,Approx->tau,
                         Approx->scale_tau);

/* initialize B-Spline knot insertion matrix, B */
init_insertion_matrix(Approx);

/* multiply B by scale_tau, store result in bscale_tau */
matrix_prod_2d(Approx->m,Approx->n,Approx->n,Approx->n,Approx->B,
            Approx->scale_tau,Approx->bscale_tau,err);
if (err != 0)
   printf("\007matrix multiplication error in approx_init()\n");

/* multiply scale_t by bscale_tau, store result in scale_b_scale */
matrix_prod_2d(Approx->m,Approx->m,Approx->m,Approx->n,Approx->scale_t,
            Approx->bscale_tau,Approx->scale_b_scale,err);
if (err != 0)
   printf("\007matrix multiplication error in approx_init()\n");

/* allocate temporary storage for matrix multiplication */
vertex_component = (float **)calloc(Approx->m,sizeof(float));
for (i = 0; i < Approx->m; i++)
   vertex_component[i] = (float *)calloc(1,sizeof(float));
vertex_product= (float **)calloc(Approx->m,sizeof(float));
for (i = 0; i < Approx->m; i++)
   vertex_product[i] = (float *)calloc(1,sizeof(float));

/* multiply scale_t by original control vertices, d */
/* for the x, y and z components */
for (xyz = 0; xyz < 3; xyz++)
  {
   for (i = 0; i < Approx->m; i++)
      vertex_component[i][0] = Approx->d[i][xyz];

   matrix_prod_2d(Approx->m,Approx->m,Approx->m,1,Approx->scale_t,
               vertex_component,vertex_product,err);

   if (err ==1)
      printf("error in approx_init calling matrix_prod_2d\n");

   /* store result and reset product matrix to zero */
   for (i = 0; i < Approx->m; i++)
     {
      Approx->scale_d[i][xyz] = vertex_product[i][0];
      vertex_product[i][0] = 0.0;
     }

  }
if (err != 0)
   printf("\007matrix multiplication error in approx_init()\n");

/* free temporary storage for matrix multiplication */
for (i = 0; i < Approx->m; i++)
  {
   free (vertex_component[i]);
   free (vertex_product[i]);
  }
free (vertex_component);
free (vertex_product);
```

```
}
/*=================================================================*
* Module Name:  init_parameters_in_approx( )
*=================================================================*
* Description:  Initializes the knot_removal list, m, n, num_to_be_removed,
*               hi_index and low_index in the Approx data structure.
*
* Input:        Pointer to the Approx data structure, pointer to the
*               correct row of the array of removal_lists found from
*               the partition, number of knot indices in the knot removal
*               list.
*
* Output:       None.
*=================================================================*/
void init_parameters_in_approx(Approx, removal_list, num_in_removal_list)
 APPROXIMATION
     *Approx;                 /* pointer to the Approx data structure */
 int
     *removal_list,         /* indices of control points to be removed */
     num_in_removal_list;         /* number of indices in removal list */
{
 int
    i;

 /* allocate Approx->removal_list */
 Approx->removal_list = (int *)calloc(num_in_removal_list,sizeof(int));

 for (i = 0; i < num_in_removal_list; i++)
   Approx->removal_list[i] = removal_list[i];

 Approx->num_to_be_removed = num_in_removal_list;

 init_hi_low_approx_indices(Approx);

 Approx->m = Approx->hi_index - Approx->low_index + 7;

 Approx->n = Approx->m - Approx->num_to_be_removed;
}
/*=================================================================*
* Module Name:  init_hi_low_approx_indices( )
*=================================================================*
* Description:  Searches for and initializes the hi and low indices of
*               control points being removed.
*
* Input:        Pointer to the Approx data structure with num_to_be_removed
*               and removal_list already initialized.
*
* Output:       None.
*=================================================================*/
void init_hi_low_approx_indices(Approx)
 APPROXIMATION
          *Approx;            /* pointer to the Approx data structure */
{
 int
    i;                      /* subscript of control point being checked */

 /* initialize hi_index and low_index */
 Approx->hi_index = Approx->low_index = Approx->removal_list[0];

 for ( i = 0; i < Approx->num_to_be_removed; i++)
   {
    if ( Approx->removal_list[i] > Approx->hi_index)
       Approx->hi_index = Approx->removal_list[i];

    if ( Approx->removal_list[i] < Approx->low_index)
       Approx->low_index = Approx->removal_list[i];
```

```
   }
}
/*====================================================================*
* Module Name:  init_vertices()
*====================================================================*
* Description:  Initializes control vertices of original curve in the
*               Approx data structure.
*
* Input:        Pointer to the component data and pointer to the Approx
*               data structure.
*
* Output:       Control vertices of original curve, Approx->d, are set
*               to the correct values for the local region of the curve.
*====================================================================*/
void init_vertices(comp, Approx, xsn)
comp_data
       *comp;                       /* pointer to the component data */
APPROXIMATION
           *Approx;                /* pointer to Approx data structure */
int
   xsn;     /* cross section number from which knots are being removed */
{
 int
   count = 0,              /* control vertex index for array to store
                                original indices                      */
   i,     /* control point number on curve in component data structure
             in the range 0...m+4-1                                   */
   xyz;                                             /* loop variable */

 for (i = Approx->low_index-1-3; i < Approx->hi_index+Approx->k-1; i++)
   {
    for (xyz = 0; xyz < 3; xyz++)
       Approx->d[count][xyz] = comp->hull[xsn][i][xyz];
    count++;
   }

}

/*====================================================================*
* Module Name:  init_knots()
*====================================================================*
* Description:  Initializes original and reduced knot sequences in Approx
*               data structure.
*
* Input:        Pointer to the component data structure, pointer to the
*               Approximate data structure, cross section number of curve
*               from which knots are being removed.
*
* Output:       Knot sequence of original curve, Approx->t, is initialized
*               to the knots of the cross section in the component data
*               structure.  Knot sequence of curve with knots removed,
*               Approx->tau, is also initialized.
*====================================================================*/
void init_knots(comp, Approx)
comp_data
      *comp;                   /* pointer to component data structure */
APPROXIMATION
           *Approx;          /* pointer to the Approx data structure */
{
 int
   count = 0,        /* knot index for array to store original indices */
   i,              /* knot number on curve in component data structure
                      in the range 0...m+4-1                          */
   j,                                    /* subscript for removal_list */
   tau_index = 0,                /* subscript for reduced knot array */
   rem_count = 0,                   /* number of knots removed from
                                       original knot sequence so far */
```

```
    not_in_removal_list = 1, /* flag to tell whether current knot is in
                                 the knot removal list                   */
    xyz;                                                /* loop variable */

 for (i = Approx->low_index-4; i < Approx->hi_index+7; i++)
   {
    Approx->t[count] = comp->w_knot[i];
    count++;
   }

 for (i = Approx->low_index-4; i < Approx->hi_index+7; i++)
   {
    not_in_removal_list = 1;
    for ( j = 0; j < Approx->num_to_be_removed; j++)
      {
       if (i == Approx->removal_list[j]+1)
         {
          /* remove knot from sequence */
          not_in_removal_list = 0;
          break;
         }
      }
    if (not_in_removal_list)
      {
       /* copy knot value from original knot sequence */
       Approx->tau[tau_index] = comp->w_knot[i];
       tau_index++;
      }
   }
}

/*********************************************************************
*    Name:  approx_alloc.c
*    Author:  Fred W. Marcaly
*    Date:  December 10, 1990
*
*    Description:  Contains modules used to allocate and free memory
*                  for the APPROXIMATION data structure.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void
     approx_alloc( ),
     approx_free( );

/*----------------------End of Function Declarations------------------*/

/*==================================================================*
* Module Name:  approx_alloc( )
*==================================================================*
* Description:  Allocates memory for the APPROXIMATION data structure
*               and sets the number of control points before and after
*               knot removal in the Approx data structure.
*
* Input:        Pointer to the Approx data structure, number of control
*               points in the original curve (m) and number of control
*               points in the new curve (n).
*
* Output:       All variables in Approx data structure are allocated.
*==================================================================*/
```

```
void approx_alloc(Approx, k)
APPROXIMATION
             *Approx;    /* pointer to the approximation data structure */
int
   k;                                              /* order of B-Splines */
{
 int
    i;                                                /* loop variable */

 Approx->k = k;    /* set new order */

 /* --------- allocate control vertices before knot removal ------------*/
 /* set up pointer to array of row pointers */
 Approx->d = (float **)calloc(Approx->m,sizeof(float *));

 /* set up pointers to rows */
 for (i = 0; i < Approx->m; i++)
  Approx->d[i] = (float *)calloc (3,sizeof(float));


 /* --------- allocate knot seqence before knot removal ----------------*/
 Approx->t = (float *)calloc(Approx->m+4,sizeof(float));


 /* --------- allocate control vertices after knot removal -------------*/
 /* set up pointer to array of row pointers */
 Approx->x = (float **)calloc(Approx->n,sizeof(float *));

 /* set up pointers to rows */
 for (i = 0; i < Approx->n; i++)
  Approx->x[i] = (float *)calloc (3,sizeof(float));


 /* --------- allocate knot seqence after knot removal ------------*/
 Approx->tau = (float *)calloc(Approx->n+4,sizeof(float));


 /* --------- allocate scaling matrix E(1/2) on knot sequence t ---------*/
 /* set up pointer to array of row pointers */
 Approx->scale_t = (float **)calloc(Approx->m,sizeof(float *));

 /* set up pointers to rows */
 for (i = 0; i < Approx->m; i++)
  Approx->scale_t[i] = (float *)calloc (Approx->m,sizeof(float));


 /* --------- allocate knot insertion matrix from t => tau ------------*/
 /* set up pointer to array of row pointers */
 Approx->B = (float **)calloc(Approx->m,sizeof(float *));

 /* set up pointers to rows */
 for (i = 0; i < Approx->m; i++)
  Approx->B [i] = (float *)calloc (Approx->n,sizeof(float));


 /* --------- allocate scaling matrix E(-1/2) on knot sequence tau -----*/
 /* set up pointer to array of row pointers */
 Approx->scale_tau = (float **)calloc(Approx->n,sizeof(float *));

 /* set up pointers to rows */
 for (i = 0; i < Approx->n; i++)
  Approx->scale_tau[i] = (float *)calloc (Approx->n,sizeof(float));


 /* --------- allocate matrix product (B) X (scale_tau) ------------*/
 /* set up pointer to array of row pointers */
 Approx->bscale_tau = (float **)calloc(Approx->m,sizeof(float *));
```

```
/* set up pointers to rows */
for (i = 0; i < Approx->m; i++)
 Approx->bscale_tau[i] = (float *)calloc (Approx->n,sizeof(float));


/* --------- allocate matrix product (scale_t) X (B) X (scale_tau) ------*/
/* set up pointer to array of row pointers */
Approx->scale_b_scale = (float **)calloc(Approx->m,sizeof(float *));

/* set up pointers to rows */
for (i = 0; i < Approx->m; i++)
   Approx->scale_b_scale[i] = (float *)calloc (Approx->n,sizeof(float));


/* --------- allocate matrix product (scale_t) X (d) ------------*/
/* set up pointer to array of row pointers */
Approx->scale_d = (float **)calloc(Approx->m,sizeof(float *));

/* set up pointers to rows */
for (i = 0; i < Approx->m; i++)
   Approx->scale_d[i] = (float *)calloc (3,sizeof(float));

/* --------- allocate matrix for lhs coefficients of normal equations ---- */
/* set up pointer to array of row pointers */
Approx->lhs_normal = (float **)calloc(Approx->n,sizeof(float *));

/* set up pointers to rows */
for (i = 0; i < Approx->n; i++)
   Approx->lhs_normal[i] = (float *)calloc(Approx->n,sizeof(float));


/* --------- allocate matrix for rhs of normal equations ----------- */
/* set up pointer to array of row pointers */
for (i = 0; i < Approx->n; i ++)
   Approx->rhs_normal = (float **)calloc(Approx->n,sizeof(float *));

/* set up pointers to rows */
for (i = 0; i < Approx->n; i++)
   Approx->rhs_normal[i] = (float *)calloc(3,sizeof(float));
}
/*=====================================================================*
* Module Name:  approx_free( )
*=====================================================================*
* Description:  Frees memory within the Approx data structure.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       All allocated variables in the Approx data structure are
*               freed.
*=====================================================================*/
void approx_free(Approx)
APPROXIMATION
            *Approx;            /* pointer to the Approx data structure */
{
 int
   m, /* number of control points in original curve before knots are
         removed                                                    */
   n,    /* number of control points in curve after knot(s) is removed */
   i;                                            /* loop variable */


 /*-------- set number of control points -------------------------------*/
 m = Approx->m;  /* original number */
 n = Approx->n;  /* reduced number  */

 /* ------- free memory for control vertices before knot removal -------*/
```

```
/* free pointers to rows */
for (i = 0; i < m; i++)
   free (Approx->d[i]);

/* free pointer to array of row pointers */
free(Approx->d);

/* ------ free memory for  knot seqence before knot removal ------------*/
free (Approx->t);


/* ------ free memory for control vertices after knot removal ----------*/
/* free pointers to rows */
for (i = 0; i < n; i++)
   free(Approx->x[i]);

/* free pointer to array of row pointers */
free (Approx->x);


/* ----- free memory for knot seqence after knot removal ---------------*/
free (Approx->tau);


/* ----- free memory for scaling matrix E(1/2) on knot sequence t ------*/
/* set up pointers to rows */
for (i = 0; i < m; i++)
   free (Approx->scale_t[i]);

/* set up pointer to array of row pointers */
free (Approx->scale_t);


/* ----- free memory for knot insertion matrix from t => tau -----------*/
/* free pointers to rows */
for (i = 0; i < m; i++)
   free (Approx->B [i]);

/* free pointer to array of row pointers */
free (Approx->B);


/* ----- free memory for scaling matrix E(-1/2) on knot vector tau -----*/
/* free pointers to rows */
for (i = 0; i < n; i++)
   free (Approx->scale_tau[i]);

/* free pointer to array of row pointers */
free (Approx->scale_tau);


/* ------- free memory for matrix product (B) X (scale_tau) ------------*/
/* free pointers to rows */
for (i = 0; i < m; i++)
   free (Approx->bscale_tau[i]);

/* free pointer to array of row pointers */
free (Approx->bscale_tau);


/* --- free memory for matrix product (scale_t) X (B) X (scale_tau) ---*/
/* free pointers to rows */
for (i = 0; i < m; i++)
   free (Approx->scale_b_scale[i]);

/* free pointer to array of row pointers */
free (Approx->scale_b_scale);
```

```
 /* --------- free memory for matrix product (scale_t) X (d) -----------*/
 /* free pointers to rows */
 for (i = 0; i < m; i++)
    free (Approx->scale_d[i]);

 /* free pointer to array of row pointers */
 free (Approx->scale_d);


 /* --------- free memory for lhs coefficients of normal equations ---- */
 /* free pointers to rows */
 for (i = 0; i < n; i++)
    free (Approx->lhs_normal[i]);

 /* free pointer to array of row pointers */
 free (Approx->lhs_normal);


 /* --------- free memory for rhs of normal equations ---------------- */
 /* free pointers to rows */
 for (i = 0; i < n; i++)
    free (Approx->rhs_normal[i]);

 /* free pointer to array of row pointers */
 free (Approx->rhs_normal);
}

/*********************************************************************
*    Name:  approx_solve.c
*    Author:  Fred W. Marcaly
*    Date:  December 21, 1990
*
*    Description:  Contains modules to solve the normal equations (least
*                  squares method) of the system of (m x n) equations
*                  used to find new control points during knot removal.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void back_solve_normal_equations( ),
     solve_last_normal_eqn( ),
     solve_normal_equations( ),
     solve_control_hull( );

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  solve_normal_equations( )
*=================================================================*
* Description:  Driver for solving the normal equations to determine
*               new control points for knot removal.  The system of
*               equations should already be reduced using Gaussian
*               elimination.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       Solution for new control points.
*=================================================================*/
void solve_normal_equations(Approx)
APPROXIMATION
```

```
            *Approx;            /* pointer to the Approx data structure */
{

 /* solve last equation in the system of normal equations */
 solve_last_normal_eqn(Approx);

 /* back solve the remaining equations in the system of normal equations */
 back_solve_normal_equations(Approx);

 /* multiply scale_tau by solution to get final control hull points */
 solve_control_hull(Approx);

}
/*====================================================================*
* Module Name:  solve_last_normal_eqn()
*====================================================================*
* Description:  Solves the last normal equation in the system which has
*               been reduced using Gaussian elimination.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       Solution to the last control point in the local area
*               which is affected by removing the current knot.
*====================================================================*/
void solve_last_normal_eqn(Approx)
APPROXIMATION
            *Approx;            /* pointer to the Approx data structure */
{
 int
    xyz;                        /* index for x, y or z component of RHS */

 for (xyz = 0; xyz < 3; xyz++)
    Approx->x[Approx->n-1][xyz] = Approx->rhs_normal[Approx->n-1][xyz]/
                          Approx->lhs_normal[Approx->n-1][Approx->n-1];

}
/*====================================================================*
* Module Name:  back_solve_normal_equations()
*====================================================================*
* Description:  Performs back substitution to solve equations n-2...0
*               in the system of n x n normal equations which have been
*               reduced using Gaussian elimination.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       Solution to control points 0...n-2 in the local area
*               which are affected by removing the current knot.
*====================================================================*/
void back_solve_normal_equations(Approx)
APPROXIMATION
            *Approx;            /* pointer to the Approx data structure */
{
 int
    xyz,                        /* index for x, y or z component of RHS */
    i,                                 /* index of unknown being solved */
    j;                                                /* column index */
 float
    sum_of_lhs_terms;        /* sum of the left hand side terms for which
                                solutions have already been found       */

 for (xyz = 0; xyz < 3; xyz++)
   {
    for (i = Approx->n-2; i >=0; i--)
      {
       sum_of_lhs_terms = 0.0;

       /* sum solved terms on left hand side of equation */
```

```
      for (j = i+1; j < Approx->n; j++)
         sum_of_lhs_terms = sum_of_lhs_terms +
                            Approx->lhs_normal[i][j]*Approx->x[j][xyz];

      /* divide (rhs - sum_of_lhs_terms) by coefficient of unknown on lhs */
      Approx->x[i][xyz] = (Approx->rhs_normal[i][xyz]-sum_of_lhs_terms)/
                          Approx->lhs_normal[i][i];
     } /* i loop */
   } /* xyz loop */
}
/*=================================================================*
* Module Name:  solve_control_hull()
*=================================================================*
* Description:  Calculates the new control points from the solution to
*               the system of normal equations.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       None.
*=================================================================*/
void solve_control_hull(Approx)
APPROXIMATION
          *Approx;          /* pointer to the Approx data structure */

{
 int
   i,                                              /* row number */
   xyz,                               /* x, y and z coordinate index */
   err = 0;                     /* error flag for matrix multiplication */
 float
   **temp,           /* temporary array for multiplying xyz coordinates
                        of solution matrix, Approx->x                 */
   **result;         /* temporary array for result of multiplication of
                        each coordinate of solution matrix by scale_tau */

 /* allocate memory for temp and result arrays */
 temp = (float **)calloc(Approx->n,sizeof(float*));
 result = (float **)calloc(Approx->n,sizeof(float *));
 for (i = 0; i < Approx->n; i++)
   {
    temp[i] = (float *)calloc(1,sizeof(float));
    result[i] = (float *)calloc(1,sizeof(float));
   }

 /* calculate each coordinate of new control hull */
 for (xyz = 0; xyz < 3; xyz++)
   {
    /* initialize result matrix */
    for ( i = 0; i < Approx->n; i++)
       result[i][0] = 0.0;

    /* load temporary matrix from solution array, Approx->x */
    for (i = 0; i < Approx->n; i++)
      {
       temp[i][0] = Approx->x[i][xyz];
      }

    /* find matrix product of Approx->scale_tau X Approx->x[i] */
    matrix_prod_2d(Approx->n,Approx->n,Approx->n,1,Approx->scale_tau,
                   temp, result, err);
    if (err == 1)
       printf("solve_control_hull()\terror in matrix multiplication\n");

    /* store result in Approx->x */
    for (i = 0; i < Approx->n; i++)
       Approx->x[i][xyz] = result[i][0];
```

```
   } /* end xyz loop */

 /* free allocated memory */
 for (i = 0; i < Approx->n; i++)
   {
    free(result[i]);
    free(temp[i]);
   }
 free ( result );
 free ( temp );
}
/*********************************************************************
*    Name:  crv_save_approx.c
*    Author:  Fred W. Marcaly
*    Date:  January 7, 1991
*
*    Description:  Contains modules to save a reduced cross section
*                  in the new model data structure created during
*                  data reduction.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*-----------------------Function Declarations-------------------------*/

void
    crv_save_approximated_cross_section( ),
    crv_make_knot_sequence( ),
    crv_save_xs_knots( ),
    crv_make_control_hull( ),
    crv_save_xs_hull( );

/*-----------------------End of Function Declarations------------------*/

/*====================================================================*
* Module Name:  crv_save_approximated_cross_section( )
*====================================================================*
* Description:  Stores the data for a reduced cross section of a
*               component in the component of the new model pointed to
*               by new_comp.
*
* Input:        Pointer to the component in the original model,
*               pointer to the component in the new model, pointer to
*               the Approx data structure and the cross section number
*               of the data being stored.
*
* Output:       None.
*====================================================================*/
void crv_save_approximated_cross_section(comp, new_comp, Approx, xsn)
comp_data
         *comp,                /* pointer to the component in the
                                  current Model which is being reduced */
         *new_comp;        /* pointer to the component in the new model
                              where the cross section data is to be
                              stored                                   */
APPROXIMATION
            *Approx;        /* pointer to the Approx data structure     */
int
   xsn;                     /* cross section number of the curve        */
{
 int
   i,                                               /* loop variable */
   size_needed;             /* amount of memory needed                  */
 float
```

```
      *knot,                    /* knot sequence for new cross section */
      **hull;                    /* control hull for new cross section */


 /* allocate memory for knot and control hull array of cross section */
 size_needed = comp->nw_knots - Approx->num_to_be_removed;
 knot = (float *)calloc(size_needed, sizeof(float));

 size_needed = size_needed - 4;
 hull = (float **)calloc(size_needed, sizeof(float *));
 for (i = 0; i < size_needed; i++)
   {
    hull[i] = (float *)calloc(3,sizeof(float));
   }

 /* construct array of knot values for entire cross section */
 crv_make_knot_sequence(comp, Approx, knot);

 /* construct array of control points for entire cross section */
 crv_make_control_hull(comp, Approx, xsn, hull);

 /* save knots for cross section */
 crv_save_xs_knots(new_comp, Approx->k, comp->nw_knots-4-comp->nw_removed,
                   knot);


 /* save control hull for cross section */
 crv_save_xs_hull(new_comp, xsn, comp->nw_knots-4-comp->nw_removed, hull);

 /* free knot and control point memory */
 free(knot);
 free(hull);
}
/*=====================================================================*
* Module Name:  crv_make_knot_sequence( )
*=====================================================================*
* Description:  Constructs the new knot sequence for a reduced curve
*               from data in the Model and Approx data strucutres.
*
* Input:        Pointer to the component being reduced, pointer to the
*               Approx data structure and pointer to the 1-d array
*               for knot values.
*
* Output:       1-d array of knot values.
*=====================================================================*/
void crv_make_knot_sequence(comp, Approx, knot)
comp_data
        *comp;                  /* pointer to component data structure */
APPROXIMATION
             *Approx;         /* pointer to the Approx data structure */
float
     *knot;                /* pointer to the 1-d array of knot values */
{
 int
    count = 0,         /* knot index for array to store original indices */
    i,               /* knot number on curve in component data structure
                        in the range 0...m+4-1                           */
    j,                                    /* subscript for removal_list */
    knot_index = 0,              /* subscript for reduced knot array */
    rem_count = 0,                 /* number of knots removed from
                                      original knot sequence so far  */
    not_in_removal_list = 1, /* flag to tell whether current knot is in
                                the knot removal list                 */
    xyz;                                           /* loop variable */

 /* loop through all knot values in original component */
 for (i = 0; i < comp->nw_knots; i++)
```

```
   {
    /* assume knot is not in the removal list */
    not_in_removal_list = 1;

    /* check whether current knot is in removal list */
    for ( j = 0; j < Approx->num_to_be_removed; j++)
      {
       if (i == Approx->removal_list[j]+1)
         {
          /* remove knot from sequence */
          not_in_removal_list = 0;
          break;
         }
      }
    if (not_in_removal_list)
      {
          /* copy knot value from original knot sequence */
          knot[knot_index] = comp->w_knot[i];
          knot_index++;
      }
   }
}
/*=====================================================================*
* Module Name:  crv_save_xs_knots( )
*=====================================================================*
* Description:  Saves the knot sequence for a reduced cross section.
*
* Input:        Pointer to the new component, order of B-Spline, number
*               of control points, new knot sequence.
*
* Output:       None.
*=====================================================================*/
void crv_save_xs_knots(comp, k, n, knot)
comp_data
         *comp;
int
   k,                                           /* order of B-Spline */
   n;                   /* number of control points in new cross section */
float
     *knot;                                      /* array of knot values */
{
 int
    i;                                                    /* knot index */
 for (i = 0; i < n+k; i++)
    comp->w_knot[i] = knot[i];
}
/*=====================================================================*
* Module Name:  crv_make_control_hull( )
*=====================================================================*
* Description:  Constructs the control hull for the new cross section of
*               a reduced component from the original component data
*               structure and the Approx data structure.
*
* Input:        Pointer to the original component data structure, pointer
*               to the Approx data structure.
*
* Output:       Control hull for cross section of new component.
*=====================================================================*/
void crv_make_control_hull(comp, Approx, xsn, hull)
comp_data
         *comp;    /* pointer to the original component data structure */
APPROXIMATION
             *Approx;          /* pointer to the Approx data structure */
int
   xsn;                                /* cross section being reduced */
float
```

```
     **hull;    /* 2-d array of control vertices for new cross section */
{
 int
   knot_is_in_removal_list,              /* flag to tell if knot is in
                                            removal list (false = 0)   */
   i,              /* current control point number on original curve */
   j,                                    /* index in knot removal list */
   new_vertex_count = 0,/* counts the number of new vertices that have
                          been put into the control hull             */
   xyz;                                      /* xyz coordinate index */


 /* loop through control points of original curve */
 for ( i = 0; i < comp->nw_knots-4-Approx->num_to_be_removed; i++)
   {
    /* assume knot is not in removal list */
    knot_is_in_removal_list = 0;

    /* check whether control vertex was removed */
    for ( j = 0; j < Approx->num_to_be_removed; j++)
      {
       if ( i == Approx->removal_list[j]-1)
         {
          knot_is_in_removal_list = 1;
          Approx->rem_index = Approx->removal_list[j];
          break;
         }
      }

    if (i < Approx->low_index-4)
      {
       for (xyz = 0; xyz < 3; xyz++)
         {
          hull[i][xyz] = comp->hull[xsn][i][xyz];
         }
      }
    else if (i >= Approx->low_index-4  &&
             i < Approx->hi_index+Approx->k-Approx->num_to_be_removed-1)
      {
       for (xyz = 0; xyz < 3; xyz++)
          hull[i][xyz] = Approx->x[new_vertex_count][xyz];
       new_vertex_count++;
       }
    else if ( i >= Approx->hi_index+Approx->k-Approx->num_to_be_removed-1)
      {
       for (xyz = 0; xyz < 3; xyz ++)
          hull[i][xyz] = comp->hull[xsn][i+Approx->num_to_be_removed][xyz];
      }
   } /* end for i loop */
}
/*====================================================================*
* Module Name:  crv_save_xs_hull()
*====================================================================*
* Description:  Saves the control hull for the new component cross
*               section in the new component data structure.
*
* Input:        Pointer to the new component, cross section number being
*               saved, number of control points in new
*               component and 3-d array of control vertices.
*
* Output:       None.
*====================================================================*/
void crv_save_xs_hull(comp, xsn, n, hull)
comp_data
        *comp;                          /* pointer to the new component */
int
   xsn,                              /* cross section number being saved */
```

```
   n;                  /* number of control points in new cross section */
float
     **hull;                          /* 3-d array of control points */
{
 int
    i,xyz;                     /* control point and coordinate indices */

 /* loop through each control point for the cross section */
 for (i = 0; i < n; i++)
   {
    for (xyz = 0; xyz < 3; xyz++)
      {
       comp->hull[xsn][i][xyz] = hull[i][xyz];
      }
   }
}

/**********************************************************************
*    Name:  insert_mat.c
*    Author:  Fred W. Marcaly
*    Date:  December 10, 1990
*
*    Description:  Contains modules to construct the B-Spline Knot
*                  insertion matrix needed for knot removal.
*
**********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void
     init_insertion_matrix();
float
     alpha();

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  init_insertion_matrix()
*=================================================================*
* Description:  Constructs the knot insertion matrix needed to obtain the
*               least squares solution of control vertices for the new
*               surface in knot removal.
*
* Input:        Pointer to the Approx data structure.
*
* Output:       B-Spline knot insertion matrix, Approx->b, initialized.
*==================================================================*/
void init_insertion_matrix(Approx)
APPROXIMATION
             *Approx;    /* pointer to the Approximation data structure */
{
int
   i,  /* column number in B-spline knot insertion matrix (i = 0...n-1) */
   j;     /* row number in B-spline knot insertion matrix (j = 0...m-1) */

 for (j = 0; j < Approx->m; j++)
   {
    for (i = 0; i < Approx->n; i++)
      {
       Approx->B[j][i] = alpha(Approx->tau, Approx->t, i, Approx->k, j);
      }
   }
```

```
}

/*********************************************************************
*    Name:  normal_eqns.c
*    Author:  Fred W. Marcaly
*    Date:  December 19, 1990
*
*    Description:  Contains modules to find the normal equations used in
*                  the least squares approach to solve a system of m
*                  equations in n unknowns.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations------------------------*/

 void
      make_normal_equations( ),
      make_rhs_normal_coeffs( ),
      make_lhs_normal_coeffs( );

/*----------------------End of Function Declarations------------------*/

/*==================================================================*
* Module Name:  make_normal_equations( )
*==================================================================*
* Description:  Finds the Right Hand Side (rhs) coefficients and Left
*               Hand Side (lhs) coefficients for the system of normal
*               equations used to solve an inconsistent system of
*               equations with m equations and n unknowns of the form:
*
*                    Original System:   (A)x = (B)
*                    Normal Equations:  (lhs)x = (rhs)
*
* Input:        Pointer to the 2-dimensional (m x n) left hand side matrix,
*               pointer to the 1-dimensional right hand side matrix,
*               number of equations, number of unknowns, pointer to the
*               2-dimensional matrix for the lhs coefficients of the
*               normal equations and pointer to the 1-dimensional matrix
*               for the rhs of the normal equations.
*
* Output:       Lhs and rhs coefficients for normal equations are
*               initialized.
*==================================================================*/
 void make_normal_equations(a, b, m, n, lhs, rhs)
 float
      **a,        /* pointer to the 2-dimensional coefficient matrix in
                     the original system of equations              */
      **b,         /* pointer to the 2-dimensional rhs matrix in the
                     original system of equations (m x 3)          */
      **lhs,      /* pointer to the 2-dimensional coefficient matrix of
                     the normal equations                          */
      **rhs;       /* pointer to the 2-dimensional rhs matrix of the
                     normal equations (m x 3)                      */
 int
    m,      /* number of equations in the original system of equations */
    n;      /* number of unknowns in the original system of equations
               as well as the number of normal equations and unknowns
               in the normal equations                             */
{
 int
     i, j;                                     /* loop variables */

 /* construct left hand side matrix for normal equations */
```

```
 make_lhs_normal_coeffs(a, m, n, lhs);

 /* construct right hand side for normal equations */
 make_rhs_normal_coeffs(a, b, m, n, rhs);

}
/*===================================================================*
* Module Name:  make_lhs_normal_coeffs( )
*====================================================================*
* Description:  Finds the left hand side (lhs) coefficients of the
*               normal equations used to solve a system of (m x n)
*               inconsistent equations using the method of least squares.
*
* Input:        Pointer to the 2-dimensional (m x n) left hand side matrix,
*               number of equations, number of unknowns, pointer to the
*               2-dimensional matrix for the lhs coefficients of the
*               normal equations.
*
* Output:       Lhs coefficients for normal equations are initialized.
*====================================================================*/
 void make_lhs_normal_coeffs(a, m, n, lhs)
 float
      **a,        /* pointer to the 2-dimensional coefficient matrix in
                     the original system of equations                 */
      **lhs;      /* pointer to the 2-dimensional coefficient matrix of
                     the normal equations                             */
 int
    m,       /* number of equations in the original system of equations */
    n;       /* number of unknowns in the original system of equations
                as well as the number of normal equations and unknowns
                in the normal equations                                */
{
 int
    i,j,k;                                       /* loop variables */

 /* take the partial with respect to each unknown of the sum of the
    squares of the residuals for the entire system of equations */

 for (i = 0; i < n; i++)
   {
    for (j = 0; j < n; j++)
      {
       /* initialize lhs to zero */
       lhs[i][j] = 0.0;

       for (k = 0; k < m; k++)
         {
          /* set left hand side coefficient */
          lhs[i][j] = lhs[i][j] + a[k][i] * a[k][j];
         }
      } /* j loop */
   } /* i loop */

}
/*===================================================================*
* Module Name:  make_rhs_normal_coeffs( )
*====================================================================*
* Description:  Finds the right hand side (rhs) coefficients of the
*               normal equations used to solve a system of (m x n)
*               inconsistent equations using the method of least squares.
*
* Input:        Pointer to the 2-dimensional (m x n) left hand side matrix,
*               pointer to the 1-dimensional right hand side matrix,
*               number of equations, number of unknowns, pointer to the
*               1-dimensional matrix for the rhs of the normal equations.
*
```

```
* Output:       Rhs coefficients for normal equations are initialized.
*===========================================================================*/
 void make_rhs_normal_coeffs(a, b, m, n, rhs)
 float
      **a,        /* pointer to the 2-dimensional coefficient matrix in
                     the original system of equations                      */
      **b,        /* pointer to the 2-dimensional rhs matrix in the
                     original system of equations (m x 3)                  */
      **rhs;      /* pointer to the 2-dimensional rhs matrix of the
                     normal equations (n x 3)                              */
 int
    m,       /* number of equations in the original system of equations */
    n;       /* number of unknowns in the original system of equations
                as well as the number of normal equations and unknowns
                in the normal equations                                    */
{
 int
    i, k, xyz;                                       /* loop variables */


 /* take the partial with respect to each unknown of the sum of the
    squares of the residuals for the entire system of equations */

 /* loop through x, y and z components */
 for (xyz = 0; xyz < 3; xyz++)
   {
    for (i = 0; i < n; i++)
      {
       /* initialize rhs to zero */
       rhs[i][xyz] = 0.0;

       for (k = 0; k < m; k++)
          rhs[i][xyz] = rhs[i][xyz] + b[k][xyz] * a[k][i];
      }
   }

}

/***************************************************************************
*    Name:  scale_matrix.c
*    Author:  Fred W. Marcaly
*    Date:  December 5, 1990
*
*    Description:  Contains modules to calculate scaling matrices E-1/p T
*                  for a given value of p and a knot seqeunce T.
*
***************************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include<math.h>

/*----------------------Function Declarations-------------------------*/

void
     construct_scaling_matrix();

/*----------------------End of Function Declarations------------------*/

/*=====================================================================*
* Module Name:  construct_scaling_matrix()
*=====================================================================*
* Description:  Constructs a square diagonal scaling matrix.
*
* Input:        Order of the B-Spline curve, dimension of dynamically
*               allocated 2-d scaling matrix (m x m), value of p and the
```

```
*               knot sequence from which the scaling matrix is to be
*               calculated.
*
* Output:       (m x m) diagonal scaling matrix.
*
* Note:         This routine assumes that p < inf, always.
*===================================================================*/
void construct_scaling_matrix(k, m, p, knot, scale_mat)
int
   k,                                            /* order of B-Spline */
   m;       /* dimension of the square scaling matrix to be constructed */
float
     p,                  /* denominator of exponent for scaling matrix */
     knot[],     /* knot sequence used to calculate elements of scaling
                    matrix                                             */
     **scale_mat;            /* double pointer to dynamically allocated
                                scaling matrix                         */
{
 int
    i, j;                                  /* row and column indices */
 float
      exponent;

 /* calculate exponent */
 if (p != 0.0)
    exponent = 1/p;
 else
   {
    exponent = 1.0;
    printf("\007Attempted division by zero in construct_scaling_matrix( ).\t");
    printf("Setting exponent of scaling matrix to 1.0\n");
    message("ATTEMPTED DIVISION BY ZERO IN CONSTRUCT_SCALING_MATRIX( )",1);
   }

 /* construct scaling matrix */
 for (i = 0; i < m; i ++)
   {
    for (j = 0; j < m; j ++)
      {
       if (i == j)
        {
         scale_mat[i][j] = pow( ((knot[i+k] - knot[i])/(float)k ), exponent );
        }
       else
        {
         scale_mat[i][j] = 0.0;
        }
      }
   }

}
```

B.3 Data Reduction for Non-Uniform Cubic B-Spline Curves

```
/**********************************************************************
*    Name:  crv_approx.c
*    Author:  Fred W. Marcaly
*    Date:  January 22, 1991
*
*    Description:  Contains modules to calculate new control vertices
*                  for a B-Spline curve.
*
**********************************************************************/

#include<afmnc.h>
#include<stdio.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations------------------------*/

void
     echo_bspline_model( ),
     crv_save_approximated_cross_section( ),
     cp_comp_attributes( );
int
     count_models( );
float
     curve_curve_distance( );
MODEL
     *crv_approximate( ),
     *delete_model( );

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  crv_approximate( )
*=================================================================*
* Description:  Main driver for calculating new control vertices.
*               Solves a set of (m x n) equations, where m >= n, to
*               determine the new set of control vertices.  Also swaps
*               the current model with the reduced model (next model).
*
* Input:        Pointer to the Model and pointer to the Approx data
*               structure.
*
* Output:       Returns pointer to the final model resulting from removing
*               all knots in the current partition.
*=================================================================*/
MODEL *crv_approximate(Initial_Model, Approx, removal_list,
                       num_in_removal_list, num_removal_lists,
                       partition_size, comp_to_be_reduced, xsn)
MODEL
     *Initial_Model;                  /* pointer to the initial Model */
APPROXIMATION
          *Approx;            /* pointer to the Approx data structure */
int
     **removal_list,                /* 5x5 array of knot removal lists */
     num_in_removal_list[ ],     /* number of knots in each removal list */
     num_removal_lists,        /* number of actual removal lists in array
                                  removal_list */
     partition_size,                /* number of knots in partition */
     comp_to_be_reduced,     /* component number of curve to be reduced */
     xsn;        /* cross section number of curve from which knots are
```

```
                  being removed                                    */
{
 int
    t,
    group = -1,                       /* number of the current removal
                                         list within a partition       */
    p,
    i,                                     /* component loop variable */
    j,                                /* subscript of removal array   */
    k = 4,                                   /* order of B-Splines */
    n_knots_rem_from_partition = 0;
 float
    max_error,            /* maximum error resulting from knot removal */
    avg_error;            /* average error resulting from knot removal */
 char
    error_message[60];          /* character string for error message */
 comp_data
    *comp,                         /* pointer to the current component */
    *prev_comp,                   /* pointer to the previous component */
    *new_comp,               /* pointer to the current component in the
                                new model                             */
    *prev_new_comp;        /* pointer to the previous component in the
                              new model                               */
 MODEL
    *Model;                         /* model currently being reduced */

 Model = Initial_Model;
 while(n_knots_rem_from_partition < partition_size)
   {
    /* set group to be removed from partition */
       group++ ;

    if (group > 0)
      {        /* first removal_list has already been removed from partition */
       /* allocate memory for next model */
       Model->next = (MODEL *)malloc(sizeof(MODEL));

       /* set up linked list pointers for new model */
       Model->next->prev = Model;
       Model->next->next = (MODEL *)NULL;

       /* copy model attributes to next model */
       cp_model_attributes(Model,Model->next);

      }

    for (p = 0; p < Model->num_comp; p++)
      {
       if (p == 0)
         {
          comp = Model->comp;
          new_comp = Model->next->comp;
         }
       else
         {
          comp = prev_comp->next;
          new_comp = new_comp->next;
         }

       if (comp->comp_number == comp_to_be_reduced)
         {
          /* initialize parameters in approx data structure */
          init_parameters_in_approx(Approx, &removal_list[group][0],
                                   num_in_removal_list[group]);

          /* allocate and copy new component into in next model */
```

```
        cp_comp_attributes(comp, new_comp);

        /* initialize number of knots to be removed from comp */
        comp->nw_removed = num_in_removal_list[group];

        /* allocate memory for B-Spline data in new model (Model->next) */

        crv_alloc_bspline_comp_memory(comp,new_comp);

        /* copy u_knots of original component to reduced component */
        cp_comp_u_knots(comp,new_comp);

        /* set number of knots being removed from component */
        comp->nu_removed = 0;
        comp->nw_removed = num_in_removal_list[group];

        /* allocate memory for the Approx data structure */
        approx_alloc(Approx, k);

        /* initialize Approx data structure */
        approx_init(comp, Approx, xsn);

        /* construct normal equations for least squares solution */
        make_normal_equations(Approx->scale_b_scale,Approx->scale_d,Approx->m,
                              Approx->n,Approx->lhs_normal,Approx->rhs_normal);

        /* perform Gaussian elimination on system of normal equations
           to find new control vertices */
        fwd_2d_Gauss_elim(Approx->n,Approx->n,Approx->lhs_normal,
                          Approx->rhs_normal,Approx->n);

        /* solve system of normal equations to find new control points */
        solve_normal_equations(Approx);

        /* save approximated cross section in next Model */
        crv_save_approximated_cross_section(comp,new_comp,Approx,xsn);

        /* calculate maximum and average error for approximating curve */
        max_error = curve_curve_distance(Approx->t,Approx->d,Approx->m+4,
                    Approx->tau,Approx->x,Approx->n+4,Approx->k,&avg_error);

        /* make and display message indicating error */
        sprintf(error_message,"MAXIMUM ERROR = %.5e",max_error);
        message(error_message,0);
        sprintf(error_message,"AVERAGE ERROR = %.5e",avg_error);
        message(error_message,1);

        /* free memory in Approx data structure */
        approx_free(Approx);
       } /* end of if comp->comp_number block */

    prev_comp = comp;
    prev_new_comp = new_comp;
    }  /* end p < Model->num_comp loop */

  /* update number of knots removed from partition */
  n_knots_rem_from_partition += num_in_removal_list[group];

  /* update linked list of models */
  if ( count_models(Initial_Model) > 2 )
    {
     /* delete model from linked list */
     Model = delete_model(Model);
    }

  /* move to next model */
  Model = Model->next;
```

```
   } /* end while */

 /* return final model pointer from this partition */
 return(Model);
}
/***********************************************************************
*    Name:    curve_main.c
*    Author:  Fred W. Marcaly
*    Date:    August 18, 1990
*
*    Description:  Main menu routine for B-Spline curve module.
*
***********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include<stdio.h>

/*----------------------Function Declarations-------------------------*/
MODEL *nub_curve();
void curve_insert();
void curve_intersect();
void curve_fillet(),
     curve_remove_knot(),
     display_hull_fwm();

/*----------------------End of Function Declarations------------------*/

/*====================================================================*
* Module Name: bspline_curve
*=====================================================================*
* Description: Main module for all B-Spline curve work.
*
* Input: Pointer to Model data file.
*
* Output: None.
*====================================================================*/

 void bspline_curve(Model)
 MODEL *Model;
 {
  int Return = 0;                          /* return code */
  int no_items = 6;                        /* Menu Parameters */
  static char *title = "CURVE";
  static char *items[] = { "RETURN",
                           "INSERT KNOT",
                           "INTERSECT",
                           "FILLET CURVES",
                           "REDUCE CURVES",
                           "CONTROL HULL"   };
  MODEL
     *Final_Model;                    /* final_model after data reduction */


  /* --- invert and display curves --- */
  Model = nub_curve(Model);

  newmenu(title,no_items,items);                          /* display menu */

  while ( Return != 1 )
    {
      Return = proc_input();

      if ( Return > 0 )
        {
          switch (Return)
```

```
        {
         case (1):                                    /* return */
                  Return = 1;
                  break;

         case (2):                                    /* knot insertion */
                   curve_insert(Model);
                   break;

         case (3):                                    /* intersection */
                   curve_intersect(Model);
                   break;

         case (4):                                    /* curve filleting */
                   curve_fillet(Model);
                   break;

         case (5):                                    /* curve reduction */
                   reduce_curve(Model);
                   break;

         case (6):                                /* display control hull*/
                   Final_Model = Model;
                   while(Final_Model->next != (MODEL *)NULL )
                      Final_Model = Final_Model->next;
                   display_hull_fwm(Final_Model);
                   break;

         default:
                   message("BAD INPUT IN B-SPLINE CURVE",1);
                   break;
        }

      }

    }

  oldmenu();
 }
/************************************************************************
*    Name:  curve_rem_knot.c
*    Author:  Fred W. Marcaly
*    Date:  February 27, 1991
*    Description:  Contains driver module for data reduction on the current
*                  model containing non-uniform B-Spline curves.
*
************************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include<stdio.h>
#include "../execs/rank.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void
    rank(),
    cp_model_attributes(),
    cp_comp(),
    cp_u_knots_to_next_model(),
    echo_bspline_model(),
    auto_zoom(),
    curve_remove_knot(),
    crv_alloc_bspline_comp_memory(),
    curve_swap_knots(),
    draw_nub_curve();
```

```
int
    find_partition_size( ),
    make_knot_removal_list( );
MODEL
    *crv_approximate( );

/*-----------------------End of Function Declarations-------------------*/

/*====================================================================*
* Module Name:  curve_remove_knot( )
*====================================================================*
* Description:  Driver module for curve knot removal routines.  Controls
*               initialization, processing and clean up routines.
*
* Input:        Pointer to the Model from which knots are to be removed,
*               number of knots to be removed from curve, cross section
*               number of curve.
*
* Output:       None.
*====================================================================*/
void curve_remove_knot(Initial_Model,num_to_be_removed,xsn)
MODEL
      *Initial_Model;            /* pointer to the MODEL from which knots
                                    are to be removed                  */
int
   num_to_be_removed,       /* number of knots to be removed from curve */
   xsn;             /* cross section from which knots are to be removed */
{
 MODEL
       *Model,
       *Final_Model = (MODEL *)NULL;  /* pointer to final reduced model */
 WEIGHT
       *Weight = (WEIGHT *)NULL; /* pointer to data structure for
                                    calculating knot removal weights    */
 APPROXIMATION
       *Approx = (APPROXIMATION *)NULL; /* pointer to data structure for
                                    approximating new control points    */

 int
    model_count = 0,
    i,
    total_num_removed = 0,      /* total number of knots removed so far */
    num_removal_lists,        /* number of groups of knots to be removed
                                 within a partition                     */
    **removal_list,                          /* array of removal lists */
    num_in_removal_list[5],     /* number of knots in each removal list */
    partition_size,        /* number of knots in partition being removed */
    comp_to_be_reduced;       /* component number of curve to be reduced */

 Model = Initial_Model;
 if (Model->nubs_root != 1)
   {
    /* swap u and w parametric curve data */
    curve_swap_knots(Model->comp);

    while (total_num_removed < num_to_be_removed)
      { /* --- remove knots one partition at a time --- */

       /* allocate memory for next model */
       Model->next = (MODEL *)malloc(sizeof(MODEL));

       /* set next model linked list pointers */
       Model->next->prev = Model;
       Model->next->next = (MODEL *)NULL;

       /* copy current Model attributes into next Model */
```

```
        cp_model_attributes(Model,Model->next);

        /* allocate data structure for calculating knot removal weights */
        Weight = (WEIGHT *)malloc(sizeof(WEIGHT));

        /* determine knot removal weights */
        weights(Model, Weight);
        free (Weight);

        /* rank knots of Model for knot removal */
        rank(Model);

        /* find partition size */
        partition_size = find_partition_size(Model,total_num_removed,
                                             num_to_be_removed);

        /* allocate array of knot removal lists */
        removal_list = (int **)calloc(5,sizeof(int *));
        for (i = 0; i < 5; i++)
           removal_list[i] = (int *)calloc(5,sizeof(int));

        /* make knot removal lists */
        num_removal_lists = make_knot_removal_list(&(Model->comp->rem[xsn][0]),
                            partition_size, total_num_removed, removal_list,
                            num_in_removal_list);

        /* allocate data structure for approximating new control points */
        Approx = (APPROXIMATION *)malloc(sizeof(APPROXIMATION));

        /* set component number to be reduced */
        comp_to_be_reduced = 1;

        /* remove knots in current partition from curve */
        Model = crv_approximate(Model, Approx, removal_list, num_in_removal_list,
                    num_removal_lists, partition_size, comp_to_be_reduced, xsn);
        free (Approx);

        total_num_removed += partition_size;

      } /* --- end while total_num_removed < num_to_be_removed --- */

    /* traverse linked list to last model */
    Final_Model = Initial_Model;
    while( Final_Model->next != (MODEL *)NULL)
      {
       model_count++;
       Final_Model = Final_Model->next;
      }
       model_count++;

    curve_swap_knots(Model->comp);


    /* display model after knot removal */
    draw_nub_curve(Final_Model);

    /* free removal_list */
    for (i = 0; i < 5; i ++)
      free(removal_list[i]);
    free(removal_list);

   }
 else
    message("NO B-SPLINE GEOMETRY FOUND",1);
}
/*==================================================================*
* Module Name:  crv_alloc_bspline_comp_memory()
```

```
*==================================================================*
* Description:  Allocates memory for B-Spline data in new model that is
*               to be used to store the reduced model.
*
* Input:        Pointer to the model which is being reduced.
*
* Output:       None.
*==================================================================*/
void crv_alloc_bspline_comp_memory(comp, new_model_comp)
 comp_data
        *comp,              /* pointer to components in original model */
        *new_model_comp;    /* pointer to components in reduced model */
{
 int
    i,j,                                          /* loop variable */
    w_size,                          /* size of memory block needed */
    u_size;                          /* size of memory block needed */

  /*--- allocate memory for new B-Spline data in reduced model ---*/
  /* allocate memory for knots in u parametric direction */
  u_size = new_model_comp->nu_knots = comp->nu_knots - comp->nu_removed;
  new_model_comp->u_knot = (float *)calloc(u_size,sizeof(float));

  /* allocate memory for knots in w parametric direction */
  w_size = new_model_comp->nw_knots = comp->nw_knots - comp->nw_removed;
  new_model_comp->w_knot = (float *)calloc(w_size,sizeof(float));

  /* allocate memory for control vertices of one curve */
  new_model_comp->hull = (float ***)calloc(1,sizeof(float **));
  new_model_comp->hull[0] = (float **)calloc(w_size-4,sizeof(float *));
  for (j = 0; j < new_model_comp->nw_knots-4; j++)
     new_model_comp->hull[0][j] = (float *)calloc(3,sizeof(float));

  /* set knot removal data array to the NULL pointer */
  new_model_comp->rem = (rem_data ***)NULL;

}
/*******************************************************************
*    Name:     read_pts.c
*    Author:   Fred W. Marcaly
*    Date:     August 13, 1990
*
*    Description: File to read in data points to be interpolated
*                 by a non-uniform B-Spline curve.
*
*******************************************************************/

#include <afmnc.h>
#include "../execs/showtime.h"
#include <stdio.h>

/*----------------------Function Declarations-----------------------*/

MODEL
     *read_pts();

/*----------------------End of Function Declarations----------------*/

/*================================================================*
* Module Name:  read_pts
*================================================================*
* Description: Reads pts to be interpolated by a non-uniform B-Spline
*              curve.
*
* Input:       Pointer to the model in which curve is to be stored and
*              inversion flag:
*                     0 -> invert
```

```
*                       1 -> do not invert.
*
* Output:      None.
*
*==========================================================================*/

 MODEL *read_pts(Model,invert)
 MODEL *Model;
 int  *invert[];
 {

   FILE *in_file;                              /* file pointer */
   comp_data *comp, *newcomp;                  /* component pointers */
   int i,j,count;

   if ( (in_file = fopen("points.dat","r")) == (FILE *)NULL )
      printf("File not found\n");
   else
     {
       /* --- Allocate memory for Curve Model --- */
       Model = (MODEL *)malloc(sizeof(MODEL));

       /* --- Set up linked list pointers for models --- */
       Model->next = (MODEL *)NULL;
       Model->prev = (MODEL *)NULL;

       /* --- Set all structure ID's to zero --- */
       Model->acs_root = -1;
       Model->nubs_root = -1;
       Model->fillet_root = -1;

       /* --- Read in number of Curves --- */
       fscanf(in_file," NUMBER OF CURVES = %d\n",&(Model->num_comp));

       for ( i = 0 ; i < Model->num_comp ; i++ )
         {
           /* --- Allocate space for new curve --- */
           newcomp = (comp_data *)malloc(sizeof(comp_data));

           if ( i == 0 )                        /* set next pointer */
             Model->comp = newcomp;
           else
             comp->next = newcomp;

           /* --- Read in component information --- */
           fscanf(in_file," NUMBER OF POINTS = %d\n",&(newcomp->acs_npts));

           /* --- Set to only one cross section --- */
           newcomp->acs_ncross = 1;

           /* --- Allocate memory for number of points --- */
           newcomp->acs_pts = (float ***)calloc(1,sizeof(float **));
           newcomp->acs_pts[0] = (float **)calloc(newcomp->acs_npts,
                                                  sizeof(float *));
           for ( j = 0 ; j < newcomp->acs_npts ; j++ )
             newcomp->acs_pts[0][j] = (float *)calloc(3,sizeof(float *));

           invert[i] = (int *)calloc(newcomp->acs_npts,sizeof(int));

           for ( j = 0 ; j < newcomp->acs_npts ; j++ )
             {
              fscanf(in_file,"%f %f %f %d\n", &(newcomp->acs_pts[0][j][0]),
                                              &(newcomp->acs_pts[0][j][1]),
                                              &(newcomp->acs_pts[0][j][2]),
                                              &(invert[i][j]) );
             }
           comp = newcomp;
```

```
        }
      comp->next = (comp_data *)NULL;
    }
 return(Model);

}
```

B.4 Data Reduction for Non-Uniform Bi-Cubic B-Spline Surfaces

```
/*************************************************************************
*    Name:  reduce.c
*    Author:  Fred W. Marcaly
*    Date:  February 27, 1991
*
*    Description:  Contains drivers for non-uniform B-Spline curve and
*                  surface data reduction.
*
*************************************************************************/

#include<afmnc.h>
#include<stdio.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations-------------------------*/

 void
    reduce_main( ),
    swap_uw( ),
    reduce_u( ),
    reduce_w( ),
    reduce_u_and_w( ),
    reduce( ),
    reduce_curve( );
 int
    get_removal_info( );
/*----------------------End of Function Declarations------------------*/

/*==================================================================*
* Module Name:  reduce_main( )
*==================================================================*
* Description:  Main menu driver for surface reduction module.
*
* Input:        Pointer to the Model.
*
* Output:       None.
*==================================================================*/
void reduce_main(Model)
 MODEL *Model;
 {
   int no_items = 4;
   int Return = 0;
   static char *title = "SURF_REDN";
   static char *items[] = { "RETURN",
                            "REDUCE U",
                            "REDUCE W",
                            "REDUCE U & W" };

   int display = 1;    /* flag to draw surfaces in reduce_u( )
                          and reduce_w( ) modules                 */

   if ( Model->nubs_root == -1 )           /* bring up B-Spline geometry */
       invert_nubs(Model);
   else
       display_nubs(Model);

   newmenu(title,no_items,items);

   while ( Return != 1 )
    {
```

```
      Return = proc_input();
      if ( Return > 0 )
       {
         switch ( Return )
           {
              case (1):                         /* Return */
                       Return = 1;
                       break;
              case (2):
                       reduce_u(Model,display);         /* Remove u knots */
                       break;
              case (3):                         /* Remove w knots */
                       reduce_w(Model,display);
                       break;
              case (4):                         /* Remove u and w knots */
                       reduce_u_and_w(Model);
                       break;
              default:
                       message("BAD INPUT IN SRF_REDUCTION",1);
                       break;
            }
         }
     }
  oldmenu();

 }
/*=====================================================================*
* Module Name:  reduce_u()
*=====================================================================*
* Description:  Driver for reducing NUB surfaces in the u direction.
*
* Input:        Pointer to the model containing the original surface.
*
* Output:       Adds a next model which contains the reduced surface.
*=====================================================================*/
 void reduce_u(Model,display)
 MODEL
       *Model;
 int
     display; /* flag indicating whether to display surface after reduction
              display = 1 --> draw surface */
{
 int
     num_to_be_removed; /* number of knots to be removed in u direction */
 float
    **dummy;
 comp_data
      *comp,
      *prev_comp;

 /* allocated memory for dummy intersection data array */
 dummy = (float **)calloc(1,sizeof(float *));
 dummy[0] = (float *)calloc(1,sizeof(float));

 /* swap u and w data for all components in Model */
 comp = Model->comp;
 while (comp != (comp_data *)NULL)
   {
    swap_uw(dummy, 0, comp);

    prev_comp = comp;
    comp = prev_comp->next;
   }

 message("ENTER NUMBER OF KNOTS TO BE REMOVED IN U DIRECTION",1);

 /* get number of knots to be removed */
```

```
  num_to_be_removed = get_removal_info( );

  /* remove knots from model */
  srf_remove_knot(Model, num_to_be_removed,display);

  /* swap u and w data for all components in Model */
  comp = Model->comp;
  while (comp != (comp_data *)NULL)
    {
     swap_uw(dummy, 0, comp);

     prev_comp = comp;
     comp = prev_comp->next;
    }

  /* swap u and w data for all components in next Model */
  comp = Model->next->comp;
  while (comp != (comp_data *)NULL)
    {
     swap_uw(dummy, 0, comp);

     prev_comp = comp;
     comp = prev_comp->next;
    }

  message("DATA REDUCTION IN U DIRECTION COMPLETE",1);

  /* free dummy memory */
  free(dummy[0]);
  free(dummy);

}
/*=====================================================================*
* Module Name:   reduce_w( )
*=====================================================================*
* Description:   Driver for reducing NUB surface in the w direction.
*
* Input:         Pointer to the model containing the original surface.
*
* Output:        Adds a next model which contains the reduced surface.
*=====================================================================*/
 void reduce_w(Model,display)
 MODEL
       *Model;
 int
     display;               /* flag to display surface after reduction
                               display = 1 --> draw surface             */
{
 int
     num_to_be_removed;  /* number of knots to be removed in w direction */

 message("ENTER NUMBER OF KNOTS TO BE REMOVED IN W DIRECTION",1);

 /* get number of knots to be removed */
 num_to_be_removed = get_removal_info( );

 /* remove knots from model */
 srf_remove_knot(Model, num_to_be_removed,display);

 message("DATA REDUCTION IN W DIRECTION COMPLETE",1);

}
/*=====================================================================*
* Module Name:   reduce_u_and_w( )
*=====================================================================*
* Description:   Driver for reducing NUB surfaces in both u and w
*                directions.
```

```
*
* Input:        Pointer to the model containing the original surface.
*
* Output:       Adds a next model which contains the reduced surface.
*=====================================================================*/
 void reduce_u_and_w(Model)
 MODEL
       *Model;
{
 int
     display = 1,             /* flag to draw surface after reduction */
     no_display = 0;      /* flag to not draw surface after reduction */

 reduce_u(Model,no_display);
 reduce_w(Model->next,display);

 message("DATA REDUCTION IN U AND W DIRECTIONS COMPLETE",1);

}
/*=====================================================================*
* Module Name:  reduce_curve( )
*=====================================================================*
* Description:  Driver for reducing NUB curves.
*
* Input:        Pointer to the model containing the original curve.
*
* Output:       Adds a next model which contains the reduced curve.
*=====================================================================*/
 void reduce_curve(Model)
 MODEL
       *Model;                    /* pointer to the model to be reduced */
{
 int
    num_to_be_removed;          /* total number of knots to be removed */

 /* get number of knots to be removed */
 num_to_be_removed = get_removal_info( );

 /* remove knots from model */
 curve_remove_knot(Model, num_to_be_removed);

}
/*=====================================================================*
* Module Name:  get_removal_info( )
*=====================================================================*
* Description:  Prompt the user to enter the number of knots to be
*               removed from the curve.
*
* Input:        None.
*
* Output:       Returns the number of knots to be removed.
*=====================================================================*/
 int get_removal_info( )
{
 int
    init = 5,                   /* default number of knots to remove */
    num_to_be_removed,
    class,                                        /* input class */
    device;                               /* input device number */
 char
    string[60];

 /* initialize string to 5 */
 sprintf(string,"%d",init);
 init_string(string);

 class = PICK;
```

```
 while ( class != STRING)
   {
    get_input(&class, &device);
    if (class == STRING)
      {
       get_string(string);
       sscanf(string,"%d",&num_to_be_removed);
      }
   }

 sprintf(string,"REMOVE %d KNOTS FROM EACH CROSS SECTION",num_to_be_removed);
 message(string,1);

 return (num_to_be_removed);

}

/**********************************************************************
*    Name:  srf_approx.c
*    Author:  Fred W. Marcaly
*    Date:  March 22, 1991
*
*    Description:  Contains modules to calculate new control vertices
*                  during surface reduction.
*
**********************************************************************/

#include<afmnc.h>
#include<stdio.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void
     echo_bspline_model( ),
     echo_approx( ),
     crv_save_approximated_cross_section( ),
     srf_alloc_bspline_comp_memory( ),
     cp_comp_attributes( );
int
     count_models( );
float
     curve_curve_distance( );
MODEL
     *srf_approximate( ),
     *delete_model( );

/*----------------------End of Function Declarations------------------*/

/*==================================================================*
* Module Name:  srf_approximate( )
*==================================================================*
* Description:  Main driver for calculating new control vertices during
*               surface reduction.
*
* Input:        Pointer to the Model and pointer to the Approx data
*               structure.
*
* Output:       Returns pointer to the final model resulting from removing
*               all knots in the current partition.
*==================================================================*/
MODEL *srf_approximate(Initial_Model, Approx, removal_list,
                       num_in_removal_list, num_removal_lists,
                       partition_size, comp_to_be_reduced)
MODEL
```

```
     *Initial_Model;                    /* pointer to the initial Model */
APPROXIMATION
           *Approx;             /* pointer to the Approx data structure */
int
     **removal_list,                   /* 5x5 array of knot removal lists */
     num_in_removal_list[],      /* number of knots in each removal list */
     num_removal_lists,        /* number of actual removal lists in array
                                  removal_list */
     partition_size,                    /* number of knots in partition */
     comp_to_be_reduced;       /* component number of curve to be reduced */

{
 int
     xsn,         /* cross section number of curve from which knots are
                     being removed                                      */
     t,
     group = -1,                       /* number of the current removal
                                          list with in a partition     */
     p,
     i,                                     /* component loop variable */
     j,                               /* subscript of removal array   */
     k = 4,                                     /* order of B-Splines */
     n_knots_rem_from_partition = 0;

 comp_data
     *comp,                         /* pointer to the current component */
     *prev_comp,                   /* pointer to the previous component */
     *new_comp,              /* pointer to the current component in the
                                new model                              */
     *prev_new_comp;       /* pointer to the previous component in the
                              new model                                */
 MODEL
     *Model;                           /* model currently being reduced */

 Model = Initial_Model;
 while(n_knots_rem_from_partition < partition_size)
   {
   /* set group to be removed from partition */
      group++ ;

   if (group > 0)
     {       /* first removal_list has already been removed from partition */
      /* allocate memory for next model */
      Model->next = (MODEL *)malloc(sizeof(MODEL));

      /* set up linked list pointers for new model */
      Model->next->prev = Model;
      Model->next->next = (MODEL *)NULL;

      /* copy model attributes to next model */
      cp_model_attributes(Model,Model->next);

     }

   for (p = 0; p < Model->num_comp; p++)
     {
      if (p == 0)
        {
         comp = Model->comp;
         new_comp = Model->next->comp;
        }
      else
        {
         comp = prev_comp->next;
         new_comp = new_comp->next;
        }
```

```
    if (comp->comp_number == comp_to_be_reduced)
      {
       /* initialize parameters in approx data structure */
       init_parameters_in_approx(Approx, &removal_list[group][0],
                                 num_in_removal_list[group]);

       /* allocate and copy new component into in next model */
       cp_comp_attributes(comp, new_comp);

       /* initialize number of knots to be removed from comp */
       comp->nw_removed = num_in_removal_list[group];

       /* allocate memory for B-spline data in new model (Model->next) */

       srf_alloc_bspline_comp_memory(comp,new_comp);


       /* copy u_knots of original component to reduced component */
       cp_comp_u_knots(comp,new_comp);

       /* set number of knots being removed from component */
       comp->nu_removed = 0;
       comp->nw_removed = num_in_removal_list[group];

       for (xsn = 0; xsn < comp->nu_knots-4; xsn++)
         {
          /* allocate memory for the Approx data structure */
          approx_alloc(Approx, k);

          /* initialize Approx data structure */
          approx_init(comp, Approx, xsn);

          /* construct normal equations for least squares solution */
          make_normal_equations(Approx->scale_b_scale,Approx->scale_d,
                Approx->m,Approx->n,Approx->lhs_normal,Approx->rhs_normal);

          /* perform Gaussian elimination on system of normal equations
             to find new control vertices */
          fwd_2d_Gauss_elim(Approx->n,Approx->n,Approx->lhs_normal,
                            Approx->rhs_normal,Approx->n);

          /* solve system of normal equations to find new control points */
          solve_normal_equations(Approx);

          /* save approximated cross section in next Model */
          crv_save_approximated_cross_section(comp,new_comp,Approx,xsn);

          /* free memory in Approx data structure */
          approx_free(Approx);
         }
      } /* end of if comp->comp_number block */

   prev_comp = comp;
   prev_new_comp = new_comp;
   }  /* end p < Model->num_comp loop */

 /* update number of knots removed from partition */
 n_knots_rem_from_partition += num_in_removal_list[group];

 /* update linked list of models */
 if ( count_models(Initial_Model) > 2 )
   {
    /* delete model from linked list */
    Model = delete_model(Model);
   }

 /* move to next model */
```

```
    Model = Model->next;

   } /* end while */

 /* return final model pointer from this partition */
 return(Model);
}
/*********************************************************************
*    Name:  srf_rem_knot.c
*    Author:  Fred W. Marcaly
*    Date:  March 27, 1991
*    Description:  Contains driver module for surface data reduction on
*                  the current model.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include<stdio.h>
#include "../execs/rank.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations------------------------*/

void
    rank( ),
    cp_model_attributes( ),
    cp_comp( ),
    cp_u_knots_to_next_model( ),
    auto_zoom( ),
    srf_remove_knot( ),
    curve_remove_knot( ),
    srf_alloc_bspline_comp_memory( ),
    curve_swap_knots( ),
    display_nubs( );

int
    find_partition_size( ),
    make_knot_removal_list( );
MODEL
    *srf_approximate( );

/*----------------------End of Function Declarations-----------------*/

/*==================================================================*
* Module Name:  srf_remove_knot( )
*==================================================================*
* Description:  Driver module for surface knot removal routines.  Controls
*               initialization, processing and clean up routines.
*
* Input:        Pointer to the Model from which knots are to be removed.
*
* Output:       None.
*===================================================================*/
void srf_remove_knot(Initial_Model,num_to_be_removed,display)
MODEL
      *Initial_Model;            /* pointer to the MODEL from which knots
                                    are to be removed                 */
int
   num_to_be_removed,       /* number of knots to be removed from curve */
   display;   /* flag to draw surface after reduction:
                 display = 1 --> draw surface
                 display = 0 --> do not draw surface                  */
{
 char temp[40];           /* temporary array for changing the model name */
 MODEL
       *Model,
```

```
      *Final_Model = (MODEL *)NULL;  /* pointer to final reduced model */
WEIGHT
      *Weight = (WEIGHT *)NULL; /* pointer to data structure for
                                   calculating knot removal weights    */
APPROXIMATION
      *Approx = (APPROXIMATION *)NULL; /* pointer to data structure for
                                   approximating new control points    */

int
   model_count = 0,
   i,
   total_num_removed = 0,      /* total number of knots removed so far */
   num_removal_lists,       /* number of groups of knots to be removed
                               within a partition                    */
   **removal_list,                        /* array of removal lists */
   num_in_removal_list[5],     /* number of knots in each removal list */
   partition_size,        /* number of knots in partition being removed */
   comp_to_be_reduced;      /* component number of curve to be reduced */
char
   progress_message[60];                       /* message for user */

Model = Initial_Model;
if (Model->nubs_root != 1)
  {
   while (total_num_removed < num_to_be_removed)
     { /* --- remove knots one partition at a time --- */

      /* allocate memory for next model */
      Model->next = (MODEL *)malloc(sizeof(MODEL));

      /* set next model linked list pointers */
      Model->next->prev = Model;
      Model->next->next = (MODEL *)NULL;

      /* copy current Model attributes into next Model */
      cp_model_attributes(Model,Model->next);

      /* allocate data structure for calculating knot removal weights */
      Weight = (WEIGHT *)malloc(sizeof(WEIGHT));

      /* determine knot removal weights */
      weights(Model, Weight);
      free (Weight);

      /* average knot removal weights in the w parametric direction */
      avg_w_comp_weights(Model);

      /* rank knots of Model for knot removal */
      rank(Model);

      /* find partition size */
      partition_size = find_partition_size(Model,total_num_removed,
                                           num_to_be_removed);

      /* allocate array of knot removal lists */
      removal_list = (int **)calloc(5,sizeof(int *));
      for (i = 0; i < 5; i++)
         removal_list[i] = (int *)calloc(5,sizeof(int));

      /* make knot removal lists */
      num_removal_lists = make_knot_removal_list(&(Model->comp->rem[0][0]),
                          partition_size, total_num_removed, removal_list,
                          num_in_removal_list);

      /* allocate data structure for approximating new control points */
      Approx = (APPROXIMATION *)malloc(sizeof(APPROXIMATION));
```

```
      /* set component number to be reduced */
      comp_to_be_reduced = 1;

      /* remove knots in current partition from curve */
      Model = srf_approximate(Model, Approx, removal_list, num_in_removal_list,
                  num_removal_lists, partition_size, comp_to_be_reduced);
      free (Approx);

      total_num_removed += partition_size;

      /* make message for user */
      sprintf(progress_message,"%d KNOTS REMOVED FROM EACH CROSS SECTION...",
              total_num_removed);
      message(progress_message,1);

     } /* --- end while total_num_removed < num_to_be_removed --- */

    /* traverse linked list to last model */
    Final_Model = Initial_Model;
    while( Final_Model->next != (MODEL *)NULL)
      {
       model_count++;
       Final_Model = Final_Model->next;
      }
       model_count++;

    /* change name of final model */
    sprintf(temp,"*%s",Final_Model->model_name);
    strncpy(Final_Model->model_name,temp,19);

    /*  auto_zoom(Final_Model, 2); */

    /* display model after knot removal */
    if (display)
      {
       draw_bspline(Final_Model);
      }

    /* free removal_list */
    for (i = 0; i < 5; i ++)
      free(removal_list[i]);
    free(removal_list);

   }
 else
    message("NO B-SPLINE GEOMETRY FOUND",1);
}
/*==================================================================*
* Module Name:  srf_alloc_bspline_comp_memory()
*===================================================================*
* Description:  Allocates memory for B-Spline data in new model that is
*               to be used to store the reduced model.
*
* Input:        Pointer to the original and reduced components.
*
* Output:       None.
*===================================================================*/
void srf_alloc_bspline_comp_memory(comp, new_model_comp)
 comp_data
        *comp,              /* pointer to components in original model */
        *new_model_comp;     /* pointer to components in reduced model */
{
 int
    xsn,
    i,j,                                          /* loop variable */
    w_size,                              /* size of memory block needed */
    u_size;                              /* size of memory block needed */
```

```
  /*--- allocate memory for new B-Spline data in reduced model ---*/
  /* allocate memory for knots in u parametric direction */
  u_size = new_model_comp->nu_knots = comp->nu_knots - comp->nu_removed;
  new_model_comp->u_knot = (float *)calloc(u_size,sizeof(float));

  /* allocate memory for knots in w parametric direction */
  w_size = new_model_comp->nw_knots = comp->nw_knots - comp->nw_removed;
  new_model_comp->w_knot = (float *)calloc(w_size,sizeof(float));

  new_model_comp->hull = (float ***)calloc(u_size-4,sizeof(float **));
  for (xsn = 0; xsn < u_size-4; xsn++)
    {
     new_model_comp->hull[xsn] = (float **)calloc(w_size-4,sizeof(float *));
     for (j = 0; j < new_model_comp->nw_knots-4; j++)
       {
        new_model_comp->hull[xsn][j] = (float *)calloc(3,sizeof(float));
       }
    }

     /* set knot removal data array to the NULL pointer */
     new_model_comp->rem = (rem_data ***)NULL;

}
```

B.5 Model Manipulation

```
/***********************************************************************
*     Name:  models.c
*     Author:  Fred W. Marcaly
*     Date:  March 1, 1991
*
*     Description:  Contains modules used to manipulate models.
*
***********************************************************************/

#include<afmnc.h>
#include<stdio.h>
#include "../execs/showtime.h"

/*-----------------------Function Declarations-------------------------*/

 void
    name_model( ),
    make_model_list( );
 int
    count_models( );
 MODEL
    *models( ),
    *change_current_model( ),
    *delete_model( );
/*-----------------------End of Function Declarations------------------*/

/*=====================================================================*
* Module Name:  models( )
*=====================================================================*
* Description:  Menu driver for manipulating models.
*
* Input:        Pointer to the current model.
*
* Output:       Returns pointer to new model if it has been changed.
*=====================================================================*/
 MODEL *models(Model)
 MODEL
     *Model;
{

  int Return = 0;                                      /* return code */
  int no_items = 5;                                /* Menu Parameters */
  static char *title = "MODELS";
  static char *items[] = { "RETURN",
                           "NEXT MODEL",
                           "PREVIOUS MODEL",
                           "LIST",
                           "NAME MODEL" };
  int number_of_models;   /* number of models in the linked list of models */
  static char
      message_string[60],                /* interactive message string */
      *window_title = "MODELS",       /* title for 2-d window */
      *model_names[20];              /* list of model names for 2-d window */
  MODEL
    *Old_Model;            /* original model passed into the "MODELS" menu */

  newmenu(title,no_items,items);          /* display new menu */

  while ( Return != 1 )                   /* loop to process input */
    {
      Return = proc_input( );

      if ( Return > 0 )
```

```
{
  switch (Return)
    {
     case (1):                                          /* Return */
              Return = 1;
              break;

     case (2):                          /* change model to next model */
              if ( Model != (MODEL *)NULL )
                {
                 message("CHANGING CURRENT MODEL TO NEXT MODEL",1);
                 Old_Model = Model;
                 Model = change_current_model(Model,1);
                 if (Model != Old_Model)
                   {
                    draw_bspline(Model);
                    sprintf(message_string,
                    "MODEL CHANGE COMPLETE.  CURRENT MODEL NAME = %s",
                    Model->model_name);
                    message(message_string,1);
                   }
                }
              else
                 message("NO GEOMETRY FILE FOUND?",1);
              break;

     case (3):                                     /* B-Spline Geometry */
              if ( Model != (MODEL *)NULL )
                {
                 message("CHANGING CURRENT MODEL TO PREVIOUS MODEL",1);
                 Old_Model = Model;
                 Model = change_current_model(Model,0);
                 if (Model != Old_Model)
                   {
                    draw_bspline(Model);
                    sprintf(message_string,
                    "MODEL CHANGE COMPLETE.  CURRENT MODEL NAME = %s",
                    Model->model_name);
                    message(message_string,1);
                   }
                }
              else
                 message("NO GEOMETRY FILE FOUND!",1);
              break;

     case (4):                                     /* list models */
              if ( Model != (MODEL *)NULL )
                {
                 make_model_list(Model,model_names);
                 number_of_models = count_models(Model);
                 std_window(window_title,number_of_models,model_names);
                }
              else
                 message("NO GEOMETRY FILE FOUND!",1);
              break;

     case (5):
              if ( Model != (MODEL *)NULL )
                 name_model(Model);
              else
                 message("NO GEOMETRY FILE FOUND!",1);
              break;

     default:
              message("BAD INPUT IN MODELS",1);
              break;
    }
```

```
        }

    }

   oldmenu();                                  /* display old menu */

  return (Model);
 }

/*=====================================================================*
* Module Name:  count_models()
*=====================================================================*
* Description:  Determines how many models exist in the linked list of
*               models.
*
* Input:        Pointer to any model in the linked list of models.
*
* Output:       Returns the number of models in the linked list.
*=====================================================================*/
 int count_models(Model)
 MODEL
      *Model;         /* Pointer to the to any model in the linked list */
{
 MODEL
      *Current_Model;
 int
    number_of_models = 0;

 Current_Model = Model;

 /* traverse linked list of models back to the first model */
 while (Current_Model->prev != (MODEL *)NULL)
   {
    Current_Model = Current_Model->prev;
   }

 /* traverse entire linked list of models and count them */
 while (Current_Model != (MODEL *)NULL)
   {
    number_of_models++;
    Current_Model = Current_Model->next;
   }

 return(number_of_models);

}
/*=====================================================================*
* Module Name:  delete_model()
*=====================================================================*
* Description:  Removes a model from the linked list of models in the
*               data structure and returns the pointer to the previous
*               model.
*
* Input:        Pointer to the Model which is to be deleted.
*
* Output:       Returns the pointer to the model preceding the model
*               which was deleted.  If the first or last model in a linked
*               list is deleted, then the pointer to the second or next
*               to last model is returned, respectively.
*=====================================================================*/
 MODEL *delete_model(Model)
 MODEL
      *Model;                       /* pointer to the model to be deleted */
{
 MODEL
      *Current_Model;
```

```
 /* remove Model from linked list */
 if (Model->prev == (MODEL *)NULL)
   {
    /* first model in linked list */
    Current_Model = Model->next;
    Current_Model->prev = (MODEL *)NULL;
   }
 else if (Model->next == (MODEL *)NULL)
   {
    /* last model in linked list */
    Current_Model = Model->prev;
    Current_Model->next = (MODEL *)NULL;
   }
 else
   {
    /* model is in the middle of the linked list */
    Current_Model = Model->prev;
    Current_Model->next = Model->next;
    Model->next->prev = Current_Model;
   }

 /* free memory of model being deleted */
 clean_model(Model);

 return (Current_Model);
}
/*==================================================================*
* Module Name:  change_current_model(Model, next)
*==================================================================*
* Description:  Changes the current model to the previous or next model.
*
* Input:        Pointer to the current model and previous or next flag.
*                      next = 0  returns previous model
*                      next = 1  returns next model
*
* Output:       New Model pointer.
*==================================================================*/
MODEL *change_current_model(Model, next)
MODEL
   *Model;                                   /* pointer to the new model */
int
   next;       /* flag to determine whether to return the previous
                  or next model:
                        next = 0  returns previous model
                        next = 1  returns next model                     */

{
 MODEL
    *New_Model;

   if (next)
     {
      if (Model->next != (MODEL *)NULL)
        {
         New_Model = Model->next;
        }
      else
        {
         message("NEXT MODEL DOES NOT EXIST",1);
         message("CURRENT MODEL UNCHANGED",1);
         New_Model = Model;
        }
     }

   else if (next == 0)
     {
```

```
      if (Model->prev != (MODEL *)NULL)
        {
         New_Model = Model->prev;
        }
      else
        {
         message("PREVIOUS MODEL DOES NOT EXIST",1);
         message("CURRENT MODEL UNCHANGED",1);
         New_Model = Model;
        }
     }

   else
      New_Model = Model;


 return (New_Model);
}
/*========================================================================*
* Module Name:  make_model_list()
*=========================================================================*
* Description:  Makes an array of the model names in the linked list of
*               models.
*
* Input:        A pointer to any model in the linked list of models whose
*               names are to be put into the array.
*
* Output:       None.
*=========================================================================*/
 void make_model_list(Model, names)
 MODEL
     *Model;
 char *names[];
{
 MODEL
      *Current_Model;
 int
     number_of_models = 0;

 Current_Model = Model;
 /* traverse linked list of models back to the first model */
 while (Current_Model->prev != (MODEL *)NULL)
   {
    Current_Model = Current_Model->prev;
   }

 /* traverse entire linked list of models and count them */
 while (Current_Model != (MODEL *)NULL)
   {
    names[number_of_models] = Current_Model->model_name;
    number_of_models++;
    Current_Model = Current_Model->next;
   }
}
/*========================================================================*
* Module Name:  name_model()
*=========================================================================*
* Description:  Allows the user to change the name of a model.
*
* Input:        Pointer to the model whose name is to be changed.
*
* Output:       None.
*=========================================================================*/
 void name_model(Model)
 MODEL
   *Model;
{
```

```
int
  class, /* input class */
  device; /* input device */
char
   name[20],
   message_string[60];

message("PLEASE ENTER MODEL NAME",1);
class = PICK;
init_string(Model->model_name);
while (class != STRING)
  {
   get_input(&class,&device);
   if (class == STRING)
     {
      get_string(name);
      strcpy(Model->model_name, name);
     }
  }

sprintf(message_string,"CURRENT MODEL NAMED %s",name);
message(message_string,1);

}
```

B.6 Utilities

```
/***********************************************************************
*    Name:  Gauss_2d.c
*    Author:  Fred W. Marcaly
*    Date:  December 20, 1990
*
*    Description:  Contains modules to perform forward Gaussian elimination
*                  on a system of m equations with n unknowns using
*                  2-dimensional matrices (arrays).
*
***********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations------------------------*/

 void
      fwd_2d_Gauss_elim( ),
      bwk_2d_Gauss_elim( );

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  fwd_2d_Gauss_elim( )
*=================================================================*
* Description:  Performs forward Gaussian elimination on a system of
*               equations up to and including the Pth equation using
*               2-dimensional arrays to represent matrices.
*
* Input:        Number of unknowns, number of equations, 2-d coefficient
*               matrix, 2-d RHS matrix and the number of the equation at
*               which to stop Gaussian elimination.
*
* Output:       Modified coefficient and RHS matrices.
*=================================================================*/
void fwd_2d_Gauss_elim(n, m, coeff, rhs, p)
 int
    n,              /* number of unknowns in the system of equations */
    m,                          /* number of equations in the system */
    p;         /* index at which to stop forward Gaussian Elimination */
 float
      **coeff,                              /* coefficient matrix (m x n) */
      **rhs;                   /* Right Hand Side (RHS) matrix (m x 3) */

{
 int
    k,          /* row and column index for subsequent eliminations */
    i,                                                /* row index */
    j,                                             /* column index */
    xyz;                   /* index for x, y or z component of RHS */
 float
      mult;                          /* multiplier for row reduction */


 for(k = 0; k<p-1; k++)
   {
    for(i = k+1; i < p;  i++)
      {
       mult = coeff[i][k] / coeff[k][k];
       for(j = k+1;  j<n;  j++)
         {
          coeff[i][j] = coeff[i][j] - mult*coeff[k][j];
         }
```

```
      coeff[i][k] = 0.0;
      for(xyz = 0; xyz < 3; xyz++)
         rhs[i][xyz] = rhs[i][xyz] - mult*rhs[k][xyz];
     }
   }

}
/*==================================================================*
* Module Name:  bwd_2d_Gauss_elim( )
*==================================================================*
* Description:  Performs backwards Gaussian elimination on a system of
*               equations back to and including the P+1th equation
*               using 2-dimensional matrices.
*
* Input:        Number of unknowns, number of equations, 2-d coefficient
*               matrix, 2-d RHS matrix and the number of the equation at
*               which to stop Gaussian Elimination.
*
* Output:       Modified coefficient and RHS matrices.
*==================================================================*/
void bwd_2d_Gauss_elim(n, m, coeff, rhs, p)
 int
    n,              /* number of unknowns in the system of equations */
    m,                        /* number of equations in the system */
    p;          /* index at which to stop forward Gaussian Elimination */
 float
      **coeff,                      /* 2-d coefficient matrix (m x n) */
      **rhs;              /* 2-d Right Hand Side (RHS) matrix (m x 3) */

{
 int
    k,            /* row and column index for subsequent eliminations */
    i,                                                 /* row index */
    j,                                              /* column index */
    xyz;                      /* index for x, y or z components of RHS */
 float
      mult;                            /* multiplier for row reduction */

 for(k = m-1; k > p-1;  k--)
   {
    for(i = k-1; i > p-1; i--)
      {
       mult = coeff[i][k-1] / coeff[k][k-1];
       for (j=m-1; j>=0; j--)
         {
          coeff[i][j] = coeff[i][j] - mult*coeff[k][j];
         }
       coeff[i][k-1] = 0.0;
       for(xyz = 0; xyz < 3; xyz++)
          rhs[i][xyz] = rhs[i][xyz] - mult*rhs[k][xyz];
      }
    }

}
/*******************************************************************
*    Name:  Gauss_elim.c
*    Author:  Fred W. Marcaly
*    Date:  October 4, 1990
*
*    Description:  Contains modules to solve for new control points used
*                  in determining the weights to rank control points of
                   the original B-Spline.
*
*******************************************************************/
```

```
#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"

/*----------------------Function Declarations-------------------------*/

 void
      solve_xpxn( ),
      fwd_Gauss_elim( ),
      bwk_Gauss_elim( );

/*----------------------End of Function Declarations------------------*/

/*===================================================================*
* Module Name:  fwd_Gauss_elim( )
*===================================================================*
* Description:  Performs forward Gaussian elimination on a system of
*               equations up to and including the Pth equation.
*
* Input:        Number of unknowns, number of equations, coefficient
*               matrix, RHS matrix and the number of the equation at
*               which to stop Gaussian Elimination.
*
* Output:       Modified coefficient and RHS matrices.
*===================================================================*/
void fwd_Gauss_elim(Weight,n, m, coeff, rhsx, rhsy, rhsz, p)
 WEIGHT
       *Weight;
 int
     n,              /* number of unknowns in the system of equations */
     m,                          /* number of equations in the system */
     p;         /* index at which to stop forward Gaussian Elimination */
 float
       coeff[ ],                                /* coefficient matrix */
       rhsx[ ],                       /* Right Hand Side (RHS) matrix */
       rhsy[ ],                       /* Right Hand Side (RHS) matrix */
       rhsz[ ];                       /* Right Hand Side (RHS) matrix */

{
 int
     k, /* row and column index for subsequent eliminations */
     i, /* row index */
     j; /* column index */
 float
       mult; /* multiplier for row reduction */

 for(k = 0; k<p-1; k++)
   {
    for(i = k+1; i < p;  i++)
      {
       mult = coeff[i*n+k] / coeff[k*n+k];
       for(j = k+1;  j<n;  j++)
         {
          coeff[i*n+j] = coeff[i*n+j] - mult*coeff[k*n+j];
         }
       coeff[i*n+k] = 0.0;
       rhsx[i] = rhsx[i] - mult*rhsx[k];
       rhsy[i] = rhsy[i] - mult*rhsy[k];
       rhsz[i] = rhsz[i] - mult*rhsz[k];
      }
   }

}
/*===================================================================*
* Module Name:  bwd_Gauss_elim( )
*===================================================================*
* Description:  Performs backwards Gaussian elimination on a system of
```

```
*               equations back to and including the P+1th equation.
*
* Input:        Number of unknowns, number of equations, coefficient
*               matrix, RHS matrix and the number of the equation at
*               which to stop Gaussian Elimination.
*
* Output:       Modified coefficient and RHS matrices.
*=====================================================================*/
void bwd_Gauss_elim(Weight,n, m, coeff, rhsx, rhsy, rhsz, p)
 WEIGHT
       *Weight;
 int
     n,                /* number of unknowns in the system of equations */
     m,                             /* number of equations in the system */
     p;          /* index at which to stop forward Gaussian Elimination */
 float
       coeff[],                                     /* coefficient matrix */
       rhsx[],                          /* Right Hand Side (RHS) matrix */
       rhsy[],                          /* Right Hand Side (RHS) matrix */
       rhsz[];                          /* Right Hand Side (RHS) matrix */

{
 int
     k, /* row and column index for subsequent eliminations */
     i, /* row index */
     j; /* column index */
 float
       mult; /* multiplier for row reduction */

 for(k = m-1; k > p-1;  k--)
   {
    for(i = k-1; i > p-1; i--)
      {
       mult = coeff[i*n+k-1] / coeff[k*n+k-1];
       for (j=m-1; j>=0; j--)
         {
          coeff[i*n+j] = coeff[i*n+j] - mult*coeff[k*n+j];
         }
       coeff[i*n+k-1] = 0.0;
       rhsx[i] = rhsx[i] - mult*rhsx[k];
       rhsy[i] = rhsy[i] - mult*rhsy[k];
       rhsz[i] = rhsz[i] - mult*rhsz[k];
      }
   }
}
/**********************************************************************
*    Name:  binary_intersect.c
*    Author:  Fred W. Marcaly
*    Date:  February 25, 1991
*
*    Description:  Contains modules to find the intersection of a line
*                  and a Non-Uniform B-Spline curve.
*
**********************************************************************/

#include<afmnc.h>
#include<math.h>
#include "../execs/showtime.h"

/*-----------------------Function Declarations--------------------------*/
void
    binary_line_curve_intersect(),
    copy_boundary(),
    absolute_sort_boundaries(),
    sort_boundaries();
int
```

```
    find_intersecting_interval();
float
    pt_vector_dist();

/*-----------------------End of Function Declarations-------------------*/

/*-----------------------Structure Definitions--------------------------*/

 typedef struct interval_boundary_type{
    float param;             /* parameter value for interval boundary */
    float point[3];                         /* point on curve at param */
    float dist;               /* distance from point on curve to line */
    } interval_boundary;

/*-----------------------End of Structure Definitions--------------------*/
/*=====================================================================*
* Module Name:  binary_line_curve_intersect()
*=====================================================================*
* Description:  Finds the intersection of a line and a NUB curve using a
*               binary search.
*
* Input:        Normal vector on the line, 3-d point at the base of the
*               normal vector, knot vector for curve, control hull for
*               curve, number of knots in the curve, order of curve.
*
* Output:       Three dimensional point of intersection within the
*               specified tolerance.
*=====================================================================*/
 void binary_line_curve_intersect(normal,knot,hull,n_knots,
                                 k,curve_point,intersection_point)
 float
      normal[],                          /* normal sequence on the line */
      curve_point[],         /* point at the base of the normal vector */
      *knot,                                           /* knot vector */
      **hull,                                         /* control hull */
      intersection_point[];                /* intersection point output */
 int
      n_knots,                         /* number of knots in the curve */
      k;                                   /* order of B-Spline curve */
{
 int
      intersecting_interval = 0, /* subscript of boundary[]->dist value where
                                  the sign of dist changes */
      i,j;
 float
      dist_interval,    /* distance between ends of interval being searched */
      interval_length,             /* length of interval being checked
                                      for intersection */
      tolerance = .01;/* maximum length of dist_interval for an intersection */
 interval_boundary
    **boundary; /* pointer to an array of three interval_boundary structs */

 /* allocate memory for boundary array */
 boundary = (interval_boundary **)calloc(3,sizeof(interval_boundary *));
 for (i = 0; i < 3; i++)
     boundary[i] = (interval_boundary *)calloc(1,sizeof(interval_boundary));

 /* set left(0), right(2) and middle(1) boundaries */
 boundary[0]->param = knot[k-1];
 boundary[2]->param = knot[n_knots - k + 1];
 interval_length = boundary[2]->param - boundary[0]->param;
 boundary[1]->param = boundary[0]->param + interval_length/2.;

 /* calculate points at boundaries */
 nub_crv_point(knot,hull,n_knots,boundary[0]->param,boundary[0]->point);
 nub_crv_point(knot,hull,n_knots,boundary[1]->param,boundary[1]->point);
 nub_crv_point(knot,hull,n_knots,boundary[2]->param,boundary[2]->point);
```

```
 /* calculate signed distances from points to line */
 boundary[0]->dist = pt_vector_dist(normal,curve_point,boundary[0]->point);
 boundary[1]->dist = pt_vector_dist(normal,curve_point,boundary[1]->point);
 boundary[2]->dist = pt_vector_dist(normal,curve_point,boundary[2]->point);

 /* sort boundaries based on distances */
 sort_boundaries(boundary);

 /* determine distance interval */
 for (i = 0; i < 2; i++)
   {
    if (boundary[i]->dist <= 0. && boundary[i+1]->dist >= 0.)
       dist_interval = boundary[i+1]->dist - boundary[i]->dist;
   }

 /* continue until search converges on intersection */
 while ( (dist_interval > tolerance ) &&
         (fabs(boundary[0]->dist) > tolerance) &&
         (fabs(boundary[1]->dist) > tolerance) &&
         (fabs(boundary[2]->dist) > tolerance)     )
   {

    /* recalculate middle, points and signed distances */
    copy_boundary(boundary[0], boundary[intersecting_interval]);
    copy_boundary(boundary[2], boundary[intersecting_interval+1]);

    interval_length = boundary[2]->param - boundary[0]->param;
    boundary[1]->param = boundary[0]->param + interval_length/2.;
    nub_crv_point(knot,hull,n_knots,boundary[1]->param,boundary[1]->point);
    boundary[1]->dist = pt_vector_dist(normal,curve_point,boundary[1]->point);

    sort_boundaries(boundary);

    /* determine new distance interval */
    for (i = 0; i < 2; i++)
      {
       if (boundary[i]->dist <= 0. && boundary[i+1]->dist >= 0.)
         {
          dist_interval = boundary[i+1]->dist - boundary[i]->dist;
         }
      }
   }

 /* sort boundary structures on the absolute value of the distance to
    get intersection stored in boundary[0] */
 absolute_sort_boundaries(boundary);

 /* set intersection point to middle point on curve */
 intersection_point[0] = boundary[0]->point[0];
 intersection_point[1] = boundary[0]->point[1];
 intersection_point[2] = boundary[0]->point[2];
}
/*=====================================================================*
* Module Name:  sort_boundaries()
*=====================================================================*
* Description:  Sorts the boundaries based on the distance between each
*               point in the boundary data structure and the line.
*
* Input:        Pointer to the array of boundary structures.
*
* Output:       Boundary array is sorted from lowest to highest distance.
*=====================================================================*/
 void sort_boundaries(boundary)
 interval_boundary
    **boundary;                        /* array of boundary structures */
{
```

```
 int
    i,j;
 interval_boundary
    *temp;

 /* since only 3 structures are being sorted, a bubble sort is used */
 for (i = 0; i < 2; i++)
   {
    for (j = 2; j > i; j --)
      {
       if (boundary[j]->dist < boundary[i]->dist )
         {
          temp = boundary[i];          /* swap structure pointers */
          boundary[i] = boundary[j];
          boundary[j] = temp;
         }
      }
   }
}
/*=====================================================================*
* Module Name:  absolute_sort_boundaries()
*=====================================================================*
* Description:  Sorts the boundaries based on the absolute value of the
*               distance between each point in the boundary data
*               structure and the line.
*
* Input:        Pointer to the array of boundary structures.
*
* Output:       Boundary array is sorted from lowest to highest distance.
*=====================================================================*/
 void absolute_sort_boundaries(boundary)
 interval_boundary
    **boundary;                        /* array of boundary structures */
{
 int
    i,j;
 interval_boundary
    *temp;

 /* since only 3 structures are being sorted, a bubble sort is used */
 for (i = 0; i < 2; i++)
   {
    for (j = 2; j > i; j --)
      {
       if (fabs((double)boundary[j]->dist) < fabs((double)boundary[i]->dist) )
         {
          temp = boundary[i];          /* swap structure pointers */
          boundary[i] = boundary[j];
          boundary[j] = temp;
         }
      }
   }
}
/*=====================================================================*
* Module Name:  find_intersecting_interval()
*=====================================================================*
* Description:  Finds the interval in which the intersection between
*               the line and the curve occurs.
*
* Input:        Pointer to the boundary data structure.
*
* Output:       Returns the subscript of the element in the array
*               boundary->dist[] that preceeds the change in sign of
*               boundary->distance.
*=====================================================================*/
int find_intersecting_interval(boundary)
 interval_boundary
```

```
    **boundary;                          /* array of boundary structures */
{
 int
    intersecting_interval = -1,
    i;
 float
    value;

 for (i = 0; i < 2; i++)
   {
    value = boundary[i]->dist * boundary[i+1]->dist;
    if (value <= 0.0)
      {
       /* distance changes sign and intersection is in this interval */
       intersecting_interval = i;
       break;
      }
   }

 /* returns -1 if no intersection exists */
 return(intersecting_interval);
}
/*=====================================================================*
* Module Name:  copy_boundary( )
*=====================================================================*
* Description:  Sets the contents of boundary_1 equal to the contents of
*               boundary_2.
*
* Input:        Pointer to the boundary data structure from which data is
*               to be copied, pointer to the boundary data structure into
*               which the data is to be copied.
*
* Output:       None.
*=====================================================================*/
void copy_boundary(boundary_1,boundary_2)
 interval_boundary
    *boundary_1,     /* pointer to boundary structure where data is to be
                        copied from */
    *boundary_2;     /* pointer to boundary structure where data is to be
                        copied to */
{
 boundary_1->param = boundary_2->param;

 boundary_1->point[0] = boundary_2->point[0];
 boundary_1->point[1] = boundary_2->point[1];
 boundary_1->point[2] = boundary_2->point[2];

 boundary_1->dist = boundary_2->dist;
}
/**********************************************************************
*    Name:   cp_comp_attr.c
*    Author: Fred W. Marcaly
*    Date:   February 28, 1991
*
*    Description: Contains modules to copy component attributes from one
*                 model to another.  No ACSYNT point data or non-uniform
*                 B-Spline data is copied.
*
**********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include <stdio.h>
#include "../subdivide/intersect.h"

/*----------------------Function Declarations-------------------------*/
```

```
void
    cp_comp_attributes();

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  cp_comp_attributes()
*=================================================================*
* Description: Copies the components of Model_1 and their attributes into
*              the components of Model_2, but does not copy ACSYNT point
*              data, NUBS data or knot removal data.
*
*        Note: It is assumed that the components in Model_2 have not yet
*              been allocated.
*
* Input:       Pointer to the model from which component attributes are
*              to be copied, pointer to the model to which component
*              attributes are to be copied.
*
* Output:      None.
*=================================================================*/
 void cp_comp_attributes(comp, new_model_comp)
  comp_data
          *comp,
          *new_model_comp;
 {
  comp_data
          *prev_comp,
          *prev_new_model_comp;

  /*--- set attributes of new_model_comp to be the same as comp ---*/
  new_model_comp->comp_number = comp->comp_number;
  strcpy(new_model_comp->comp_name,comp->comp_name);
  new_model_comp->color = comp->color;
  new_model_comp->existance = comp->existance;
  new_model_comp->nu = comp->nu;
  new_model_comp->nw = comp->nw;

 }
/*******************************************************************
*    Name:  cp_comp_u_knots.c
*    Author:  Fred W. Marcaly
*    Date:  March 1, 1991
*
*    Description:  Contains module to copy the u knots of a component.
*
*******************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include<stdio.h>
#include "../execs/rank.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations-------------------------*/

void
    cp_comp_u_knots();

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  cp_u_knots_to_next_model()
*=================================================================*
* Description:  Copies the knot values in the u parametric direction
*               from the component #1 to component #2.
*
```

```
* Input:        Pointer to the component from which u knots are to be
*               copied and pointer to the component to which u knots are
*               to be copied.
*
* Output:       None.
*==================================================================*/
void cp_comp_u_knots(comp_1,comp_2)
 comp_data
     *comp_1,          /* component from which u knots are to be copied */
     *comp_2;            /* component to which u_knots are to be copied */
{
 int
   i;

  for (i = 0; i < comp_1->nu_knots; i++)
     comp_2->u_knot[i] = comp_1->u_knot[i];

}
/*******************************************************************
*    Name:   cp_mod_attr.c
*    Author: Fred W. Marcaly
*    Date:   January 6, 1991
*
*    Description: Copies attributes of Model_1 to Model_2, but no ACSYNT
*                 point data, NUB data or knot removal data is copied.
*
*******************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include <stdio.h>
#include "../subdivide/intersect.h"

/*----------------------Function Declarations-------------------------*/

void
    cp_model_attributes();

/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  cp_model_attributes()
*=================================================================*
* Description: Copies the attributes of Model_1 into the Model_2.
*              Allocates memory in Model_2 for the same number of
*              components that are in Model_1.  Does not copy ACSYNT
*              point data or NUBS data.
*
* Input:       Pointer to the Model from which attributes and components
*              are to be copied and pointer to the Model to which
*              attributes and components are to be copied.
*
* Output:      None.
*==================================================================*/
 void cp_model_attributes(Model_1,Model_2)
 MODEL
     *Model_1,                /* model from which attributes are copied */
     *Model_2;                  /* model to which attributes are copied */
 {
  comp_data
          *comp,
          *prev_comp,
          *new_model_comp,
          *prev_new_model_comp;


  /* --- initialize Model intersection data to null --- */
```

```
  Model_2->intlist = (intersection *)NULL;

  /* --- initialize structure id's to -1 --- */
  Model_2->acs_root = -1;
  Model_2->nubs_root = -1;
  Model_2->fillet_root = -1;
  Model_2->int_root = -1;

  /* set new model name */
  strcpy(Model_2->model_name,Model_1->model_name);

  /* copy components */
  comp = Model_1->comp;
  while ( comp != (comp_data *)NULL )
        {
         /* allocate memory for a new component in Model_2 */
         new_model_comp = (comp_data *)malloc(sizeof(comp_data));

         if (comp == Model_1->comp)
           {
            /* set Model->comp pointer in Model_2 */
            Model_2->comp = new_model_comp;
           }
         else
           {
            /* set comp->next pointer in MODEL_2 */
            prev_new_model_comp->next = new_model_comp;
           }

         /* reset previous new model component pointer */
         prev_new_model_comp = new_model_comp;

         /* goto next component in current Model */
         prev_comp = comp;
         comp = prev_comp->next;
        }  /* end while */

  Model_2->num_comp = Model_1->num_comp; /* set number of components */

  /* set last next component pointer to null in Model_2 */
  prev_new_model_comp->next = (comp_data  *)NULL;

}
/*************************************************************************
*    Name:  crv_crv_dist.c
*    Author:  Fred W. Marcaly
*    Date:  February 23, 1991
*
*    Description:  Contains modules to find the maximum and average
*                  distance between two non-uniform B-Spline curves.
*
*************************************************************************/

#include<afmnc.h>
#include<math.h>
#include "../execs/showtime.h"

/*----------------------Function Declarations-------------------------*/

void
     nub_crv_point( ),
     binary_line_curve_intersect( ),
     line_curve_intersection( ),
     normal_to_nub_curve( ),
     nub_crv_points( );
int
```

```
    knot_range( );
float
    pt_vector_dist( ),
    vector_mag( );
/*----------------------End of Function Declarations------------------*/

/*=================================================================*
* Module Name:  curve_curve_distance( )
*=================================================================*
* Description:  Main driver for determining the maximum and average
*               distance between two NUB curves in the same plane.
*
* Input:        Knot vector for curve #1, control hull for curve #1,
*               number of knots for curve #1, knot vector for curve #2,
*               controll hull for curve #2, number of knots for curve #2.
*
* Output:       Maximum and average distance between the NUB curves.
*=================================================================*/
float curve_curve_distance(knot1,hull1,n_knots1,knot2,hull2,n_knots2,
                           k,avg_dist)
 int
    n_knots2,                         /* number of knots in curve #2 */
    n_knots1,                         /* number of knots in curve #1 */
    k;                                        /* order of NUB curve */
 float
    *knot1,                              /* knot vector for curve #1 */
    **hull1,                            /* control hull for curve #1 */
    *knot2,                              /* knot vector for curve #2 */
    **hull2,                            /* control hull for curve #2 */
    *avg_dist;      /* pointer to the average distance between curves */
{
 int
    i,                                   /* point number on curve #2 */
    xyz,                                    /* x, y or z coordinate */
    number_of_segments,            /* number of segments in curve #2 */
    total_num_of_pts,     /* total number of points calc'd on curve #2 */
    pts_per_seg = 8;  /* number of points checked on each curve segment
                         of curve #2                                 */
 float
    intersection_point[3],
    dist_vector[3],       /* vector defining distance from curve #1 to
                             curve #2                                */
    dist,
    param_value,      /* parameter value for finding point on curve #1 */
    crv1_point[3],                              /* point on curve #1 */
    crv2_point[3],                              /* point on curve #2 */
    crv2_next_point[3],                         /* point on curve #2 */
    unit_normal[3],          /* unit normal perpindicular to curve #2 */
    sum_of_dist,                   /* sum of distances between curves */
    max_dist = 0.,                 /* maximum distance between curves */
    **pt_list2;                         /* list of points on curve #2 */

 /* allocate memory for points on curve #2 */
 number_of_segments = n_knots2-k-3;
 total_num_of_pts = number_of_segments * pts_per_seg;

 pt_list2 = (float **)calloc(total_num_of_pts,sizeof(float *));
 for (i = 0; i < total_num_of_pts; i++)
   pt_list2[i] = (float *)calloc(3,sizeof(float));

 /* calculate points on curve #2 */
 nub_crv_points(knot2,hull2,n_knots2,k,pts_per_seg,pt_list2);

 /* determine parameter value to calculate a random point on curve #1 */
 param_value = 0.67 * (knot1[n_knots1-k] - knot1[k-1]) + knot1[k-1];

 /* calculate a single point on curve #1, point C (order must be 4) */
```

```
 nub_crv_point(knot1,hull1,n_knots1,param_value,crv1_point);

 /*--- find distance between each point on curve #2 and curve #1 ---*/
 /*---          determine maximum and average distances          ---*/
 for (i = 1; i < total_num_of_pts-2*pts_per_seg; i++)
   {
    for ( xyz = 0; xyz < 3; xyz++)
      {
       crv2_point[xyz] = pt_list2[i][xyz];
       crv2_next_point[xyz] = pt_list2[i+1][xyz];
      }

    normal_to_nub_curve(crv2_point,crv2_next_point,crv1_point,unit_normal);

    /* find the intersection of the scaled normal and curve #1 */
    binary_line_curve_intersect(unit_normal,knot1,hull1,n_knots1,k,
                                crv2_point,intersection_point);

    /* find vector from point on curve #2 to intersection point */
    for ( xyz = 0; xyz < 3; xyz++)
      dist_vector[xyz] = crv2_point[xyz] - intersection_point[xyz];

    /* find vector magnitude */
    dist = vector_mag(dist_vector);
    dist = (float)fabs((double)dist);

    sum_of_dist += dist;
    if (dist > max_dist)
       max_dist = dist;
   }

 *avg_dist = sum_of_dist / (total_num_of_pts-1);

 return(max_dist);
}
/*======================================================================*
* Module Name:   nub_crv_points()
*=======================================================================*
* Description:   Calculates points on a non-uniform B-Spline curve using
*                a specified number of points per curve segment.
*
* Input:         Knot vector, control hull, order of NUB, number of knot
*                values in knot vector and step in parameter value.
*
* Output:        Two dimensional array of points on the NUB.
*======================================================================*/
void nub_crv_points(knot,hull,n_knots,k,points_per_seg,point_list)
 float
     *knot,                                             /* knot vector */
     **hull,                          /* control hull (hull[point#][xyz]) */
     **point_list;              /* points on NUB (point_list[point#][xyz]) */
 int
    k,                                                  /* order of NUB */
    n_knots,                          /* number of knots in the knot vector */
    points_per_seg;             /* number of points in each curve segment */
{
 int
   xyz,                                         /* x, y or z coordinate */
   i,                     /* curve segement number between control points */
   j;                             /* point number with in a curve segment */
 float
   step,                 /* change in parameter value from point to point
                            being calculated                            */
   u,                                                /* parameter value */
   bf[4],                                   /* blending function values */
   tol = 0.0000000000001;
```

```
 for (i = 1; i <= n_knots-k-3; i++)    /* for each curve segement */
   {
    step = (knot[i+k-1]-knot[i+k-2])/points_per_seg;
    for (j = 0; j < points_per_seg; j++)       /* for each point in a */
      {                                         /* curve segment */
       /* set parameter value */
       u = knot[i+k-2] + ((i-1)*points_per_seg + j)*step - tol;
       blend(i+k-2,knot,u,bf);          /* calculate blending functions */

       for (xyz = 0; xyz < 3; xyz++)
         {
          point_list[(i-1)*points_per_seg + j][xyz] =
             bf[0] * hull[i-1][xyz] +
             bf[1] * hull[i][xyz] +
             bf[2] * hull[i+1][xyz] +
             bf[3] * hull[i+2][xyz];
         }  /* end xyz loop */
      } /* end j loop */
   } /* end i loop */
}
/*=====================================================================*
* Module Name:  nub_crv_point()
*=====================================================================*
* Description:  Calculates a single point on a non-uniform B-Spline curve
*               at a specified parameter value.
*        Note:  The order of the NUB curve is assumed to be 4.
*
* Input:        Knot vector, control hull, number of knot values in knot
*               vector, order of NUB and parameter value.
*
* Output:       One dimensional point array.
*=====================================================================*/
void nub_crv_point(knot,hull,n_knots,param_val,point)
 float
     *knot,                                              /* knot vector */
     **hull,                      /* control hull (hull[point#][xyz]) */
     point[],               /* points on NUB (point_list[point#][xyz]) */
     param_val;          /* parameter value for point being calculated */
 int
    n_knots;                       /* number of knots in the knot vector */
{
 int
   i,
   param_range,        /* index of knot value at the beginning of the
                   curve segment containing the point being calculated */
   xyz;                                     /* x, y or z coordinate */
 float
   u,                                             /* parameter value */
   bf[4],                                  /* blending function values */
   tol = 0.0000000000001;

   param_range = knot_range(param_val,knot,n_knots);
   blend(param_range,knot,param_val,bf);   /* calculate blending functions */

   for (xyz = 0; xyz < 3; xyz++)                  /* calculate point */
     {
      point[xyz] = bf[0] * hull[param_range-3][xyz] +
                   bf[1] * hull[param_range-2][xyz] +
                   bf[2] * hull[param_range-1][xyz] +
                   bf[3] * hull[param_range][xyz];
     }
}
/*=====================================================================*
* Module Name:  normal_to_nub_curve
*=====================================================================*
* Description:  Finds a normal to a planar NUB curve.  The normal is in
*               the plane containing the NUB curve.
```

```
*
* Input:        Three dimensional point on NUB defining the tail of a
*               vector on the NUB, 3-d point on a NUB defining the head
*               of the same vector on the NUB and another point in the
*               plane containing the NUB curve.
*
* Output:       Unit vector normal to the NUB curve at the specified point.
*=====================================================================*/
void normal_to_nub_curve(nub_crv_point1,nub_crv_point2,plane_point,unit_normal)
 float
      nub_crv_point1[],                              /* point on the NUB */
      nub_crv_point2[],                              /* point on the NUB */
      plane_point[],                         /* point in the NUB's plane */
      unit_normal[];                              /* unit normal vector */
{
 int
      xyz;                                    /* x, y and z coordinate */
 float
      nub_vector[3],                              /* vector on NUB curve */
      plane_vector[3],                        /* vector in the NUB plane */
      normal_to_curve[3],                        /* normal to the curve */
      normal_to_plane[3];                 /* vector normal to NUB plane */

 /* calculate two vectors in the plane with their tails at the point on
    on the NUB */
    for (xyz = 0; xyz < 3; xyz++)
      {
       nub_vector[xyz] = nub_crv_point2[xyz] - nub_crv_point1[xyz];
       plane_vector[xyz] = plane_point[xyz] - nub_crv_point1[xyz];
      }

 /* find a normal to the plane containing the NUB at the specified point
    on the NUB */
 cross_product(nub_vector,plane_vector,normal_to_plane);

 /* find the normal to the NUB curve */
 cross_product(normal_to_plane, nub_vector, normal_to_curve);

 /* find the unit vector for the normal to the curve */
 unit_vector(normal_to_curve, unit_normal);

}
/*=====================================================================*
* Module Name:  pt_vector_dist()
*=====================================================================*
* Description:  Calculates the perpindicular distance between a point and a
*               line defined by a 3-dimensional unit vector.
*
* Input:        Three dimensional unit vector, point on the line at the
*               tail of the unit vector and a point off the line.
*
* Output:       Returns minimum distance between point off the line and
*               the line.
*=====================================================================*/
float pt_vector_dist(unit_vector, point_on_line, point_off_line)
 float
      unit_vector[3],
      point_off_line[3],
      point_on_line[3];
{
 float
      cross[3],                                /* cross product vector */
      point_vector[3];   /* vector from point on line to point off line */

 /* find vector from point on line to point off line */
 point_vector[0] = point_off_line[0] - point_on_line[0];
 point_vector[1] = point_off_line[1] - point_on_line[1];
```

```
 point_vector[2] = point_off_line[2] - point_on_line[2];

 cross_product(point_vector, unit_vector, cross);

 return(cross[2]);
}
/***********************************************************************
*    Name:   curve_swap.c
*    Author: Fred W. Marcaly
*    Date:   January 21, 1991
*
*    Description: Contains modules to swap u and w parameter values for
*                 a NUBS curve which is stored in a model using the
*                 NUBS curve module.
*
***********************************************************************/

#include <afmnc.h>
#include <stdio.h>
#include "../execs/showtime.h"
#include "../execs/show_math.h"
#include "../subdivide/intersect.h"
#include "../ldfillet/fillet.h"

/*----------------------Function Declarations-------------------------*/
comp_data *copy_data();
void copy_intdata();
void draw_nubs();
/*----------------------End of Function Declarations------------------*/

/*===================================================================*
* Module Name:  curve_swap_knots()
*===================================================================*
* Description:  Swaps the u and w knots for a curve.
*
* Input:        Pointer to the component containing only a curve.
*
* Output:       None.
*===================================================================*/
 void curve_swap_knots(comp)
 comp_data *comp;
 {
  int i, j , xyz;                                /* looping variables */
  int count;
  float ***newhull;                                /* new control hull */
  float **newint;                             /* new intersection data */
  float *new_u, *new_w;                           /* new u and w knots */
  float tmp;                              /* tmp variable for swapping */

  new_u = (float *)calloc(comp->nw_knots,sizeof(float));
  new_w = (float *)calloc(comp->nu_knots,sizeof(float));

  /* --- swap knot values --- */
  for ( i = 0 ; i < comp->nu_knots ; i++ )
     new_w[i] = comp->u_knot[i];

  for ( i = 0 ; i < comp->nw_knots ; i++ )
     new_u[i] = comp->w_knot[i];

  /* --- swap number of u and w knots --- */
  tmp = comp->nu_knots;
  comp->nu_knots = comp->nw_knots;
  comp->nw_knots = tmp;

  free(comp->u_knot);
  free(comp->w_knot);
```

```
  comp->u_knot = new_u;
  comp->w_knot = new_w;

  /* swap open and closed flags */
  tmp = comp->open[0];
  comp->open[0] = comp->open[1];
  comp->open[1] = tmp;

 }
/*********************************************************************
*    Name:  discrete_bspline.c
*    Author:  Fred W. Marcaly
*    Date:  December 8, 1990
*
*    Description:  Contains modules to calculate descrete B-Splines.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/approximate.h"

/*----------------------Function Declarations------------------------*/

float
     alpha();

/*----------------------End of Function Declarations-----------------*/

/*==================================================================*
* Module Name:  alpha()
*==================================================================*
* Description:  Recursively calculates the discrete B-Splines needed for
*               the knot insertion matrix of order k.
*
* Input:        Pointer to the original knot sequence (tau), pointer to
*               the refined knot sequence (t), order of B-Spline (k),
*               row index of element of transformation matrix being found,
*               column index of element of transformation matrix being
*               found.
*
* Output:       Discrete B-Spline value needed to construct a knot
*               insertion matrix.
*==================================================================*/
float alpha(tau, t, i, k, j)
 float
       tau[],                               /* original knot sequence */
       t[];                  /* refined knot sequence (with knot added) */
 int
     k,                                         /* order of B-Spline */
     i,                       /* row index of element in knot insertion
                                 matrix being calculated            */
     j;                    /* column index of element in knot insertion
                              matrix being calculated               */
{
 int
    r;                                             /* loop variable */
 float
       denom1,                          /* denominator of first term */
       denom2,                         /* denominator of second term */
       value;                 /* value of discrete B-spline being found */

 if (k != 1)
   {
    denom1 = tau[i+k-1]-tau[i];
    denom2 = tau[i+k]-tau[i+1];
    if ( denom1 == 0.0 && denom2 == 0.0)
```

```
      value = 0.0;
    else if (denom1 == 0.0)
      value = ((tau[i+k] - t[j+k-1])/denom2)*alpha(tau, t, i+1, k-1, j);
    else if (denom2 == 0.0)
      value = ((t[j+k-1] - tau[i])/denom1)*alpha(tau, t, i, k-1, j);
    else
      value = ((t[j+k-1] - tau[i])/denom1)*alpha(tau, t, i, k-1, j)
           +((tau[i+k] - t[j+k-1])/denom2)*alpha(tau, t, i+1, k-1, j);
   }
 else   /* k = 1 */
    if (tau[i] <= t[j] && t[j] < tau[i+1])
      value = 1.;
    else
      value = 0.;

 return (value);
}

/*********************************************************************
*    Name:    math.c
*    Author:  Robert W. Jones & Fred W. Marcaly
*    Date:    September 12, 1990
*
*    Description: Contains all math uitility routines.
*
*********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"
#include "../execs/rank.h"
#include <math.h>

/*-----------------------Function Declarations-------------------------*/
void
     scale_vector( ),
     quicksort( ),
     matrix_prod( ),
     matrix_sub( );
float
     max_norm( ),
     vector_mag( );

/*-----------------------End of Function Declarations------------------*/

/*==================================================================*
* Module Name:  unit_vector
*===================================================================*
* Description:  Calculates the unit vector of a vector.
*
* Input:        Vector array.
*
* Output:       Unit vector.
*===================================================================*/

 void unit_vector(vector,unit_vector)
 float vector[3], unit_vector[3];
 {
  int xyz;                                /* looping variable */
  float mag;                              /* magnitude of unit vector */

  mag = vector_mag(vector);

  for ( xyz = 0 ; xyz <= 2 ; xyz++ )
      unit_vector[xyz] = vector[xyz]/mag;
 }
/*==================================================================*
* Module Name:  scale_vector( )
```

```
*======================================================================*
* Description:  Scales a 3-d vector by a specified quantity.
*
* Input:        Vector to be scaled, scaling factor.
*
* Output:       Scaled vector.
*======================================================================*/

 void scale_vector(vector, scale)
 float
      vector[3],                                /* vector to be scaled */
      scale;                                         /* scaling factor */
 {
  vector[0] = scale * vector[0];
  vector[1] = scale * vector[1];
  vector[2] = scale * vector[2];

 }
/*======================================================================*
* Module Name: vector_mag
*======================================================================*
* Description:  Calculates the magnitude of a vector.
*
* Input:        Vector array.
*
* Output:       Magnitude of vector.
*======================================================================*/

 float vector_mag(vector)
 float vector[];
 {
  float mag;                                /* magnitude of vector */

  mag = sqrt(vector[0]*vector[0] + vector[1]*vector[1] + vector[2]*vector[2]);

  return(mag);

 }

/*======================================================================*
* Module Name: cross_product
*======================================================================*
* Description:  Calculates the cross product of two vectors.
*
* Input:        Arrays containing both vectors.
*
* Output:       Array containing cross product of the vectors.
*======================================================================*/

 void cross_product(vect_1,vect_2,cross)
 float vect_1[], vect_2[], cross[];
 {

  cross[0] = vect_1[1]*vect_2[2] - vect_1[2]*vect_2[1];
  cross[1] = vect_1[2]*vect_2[0] - vect_1[0]*vect_2[2];
  cross[2] = vect_1[0]*vect_2[1] - vect_1[1]*vect_2[0];

 }

/*======================================================================*
* Module Name: dot_product
*======================================================================*
* Description:  Calculates the dot product of two vectors.
*
* Input:        Arrays containing both vectors.
*
```

```
* Output:       Dot product of both vectors.
*=====================================================================*/

 float dot_product(vect_1,vect_2)
 float vect_1[], vect_2[];
 {
  float product;                         /* dot product of both vectors */

  product = vect_1[0]*vect_2[0] + vect_1[1]*vect_2[1] + vect_1[2]*vect_2[2];

  return(product);
 }

/*====================================================================*
* Module Name: pt_pt_dist
*====================================================================*
* Description:  Calculates the distance between two points.
*
* Input:        Arrays for each point.
*
* Output:       Distance between points.
*=====================================================================*/

 float pt_pt_dist(p1,p2)
 float p1[], p2[];
 {
  float dist;                     /* distance between points */

  dist = sqrt( (p1[0] - p2[0])*(p1[0] - p2[0]) +
               (p1[1] - p2[1])*(p1[1] - p2[1]) +
               (p1[2] - p2[2])*(p1[2] - p2[2]) );

  return(dist);
 }
/*====================================================================*
* Module Name:  matrix_prod
*====================================================================*
* Description:  Finds the product of two matrices, each contained in a
*               1-d array.
*
* Input:        Number of rows in matrix 1, number of columns in matrix 1,
*               number of rows in matrix 2, number of columns in matrix 2,
*               matrix 1, matrix 2, correctly dimensioned result matrix.
*
* Output:       Result matrix and error flag.
*=====================================================================*/
 void matrix_prod(nrow_1,ncol_1,nrow_2,ncol_2,mat1,mat2,result,err)
 int
     nrow_1,                            /* number of rows in matrix 1 */
     ncol_1,                         /* number of columns in matrix 1 */
     nrow_2,                            /* number of rows in matrix 2 */
     ncol_2,                         /* number of columns in matrix 2 */
     err;                     /* error indicator if ncol_1 != nrow_2 */
 float
     mat1[],                         /* first matrix to be multiplied */
     mat2[],                        /* second matrix to be multiplied */
     result[];                /* result of the matrix multiplication */
{

 int
    i, /* column # in matrix 1; row # in matrix 2 */
    j, /* row # in matrix 1 */
    k; /* column # in matrix 2 */

 if (ncol_1 == nrow_2)
  {
    err = 0;
```

```
    for (k = 0; k < ncol_2; k++)
      {
       for (j = 0; j < nrow_1; j++)
         {
          for (i = 0; i < ncol_1; i++)
            {
             result[j*ncol_2 + k] = result[j*ncol_2 + k] +
             mat1[j*ncol_1 + i] * mat2[i*ncol_2 + k];
            }
         }
      }
   } /* end if */
 else
    err = 1;

}

/*=====================================================================*
* Module Name:  matrix_prod_2d
*=====================================================================*
* Description:  Finds the product of two matrices, each contained in a
*               2-d array.
*
* Input:        Number of rows in matrix 1, number of columns in matrix 1,
*               number of rows in matrix 2, number of columns in matrix 2,
*               matrix 1, matrix 2, correctly dimensioned result matrix.
*
* Output:       2-d result matrix and error flag.
*=====================================================================*/
 void matrix_prod_2d(nrow_1,ncol_1,nrow_2,ncol_2,mat1,mat2,result,err)
 int
     nrow_1,                              /* number of rows in matrix 1 */
     ncol_1,                           /* number of columns in matrix 1 */
     nrow_2,                              /* number of rows in matrix 2 */
     ncol_2,                           /* number of columns in matrix 2 */
     err;                        /* error indicator if ncol_1 != nrow_2 */
 float
     **mat1,                             /* first matrix to be multiplied */
     **mat2,                            /* second matrix to be multiplied */
     **result;                       /* result of the matrix multiplication */
{

 int
    i,                    /* column # in matrix 1; row # in matrix 2 */
    j,                                         /* row # in matrix 1 */
    k;                                      /* column # in matrix 2 */

 if (ncol_1 == nrow_2)
   {
    err = 0;
    for (k = 0; k < ncol_2; k++)
      {
       for (j = 0; j < nrow_1; j++)
         {
          for (i = 0; i < ncol_1; i++)
            {
             result[j][k] = result[j][k] +
             mat1[j][i] * mat2[i][k];
            }
         }
      }
   } /* end if */
 else
    err = 1;

}
```

```
/*=====================================================================*
* Module Name:  matrix_sub( )
*=====================================================================*
* Description:  Finds the difference of two matrices.
*
* Input:        Number of rows in matrices 1 and 2, number of columns in
*               matrices 1 and 2, matrix 1, matrix 2, correctly
*               dimensioned result matrix.
*
* Output:       Result matrix.
*=====================================================================*/
 void matrix_sub(nrow,ncol,mat1,mat2,result)
 int
     nrow,                          /* number of rows in matrix 1 */
     ncol;                       /* number of columns in matrix 1 */
 float
     mat1[],                                      /* first matrix */
     mat2[],                                     /* second matrix */
     result[];                /* result of the matrix subtraction */
{
 int
     i,                                              /* row index */
     j;                                           /* column index */

 for (i = 0; i < nrow; i++)
   {
    for (j = 0; j < ncol; j++)
      {
       result[i*ncol+j] = mat1[i*ncol+j] - mat2[i*ncol+j];
      }
   }
}

/*=====================================================================*
* Module Name:  max_norm( )
*=====================================================================*
* Description:  Finds the max norm for a matrix.
*
* Input:        Size of the matrix whose max norm is to be found, one
*               dimensional matrix of values.
*
* Output:       Maximum value in the one dimensional matrix.
*=====================================================================*/
 float max_norm(Weight,size, A)
 WEIGHT
       *Weight;
 int
      size;               /* size of the matrix. ie: A[0..size-1] */
 float
      A[];                 /* matrix whose max norm is to be found */
{
 int
       i;                                        /* loop variable */
 float
       value,                             /* value to be returned */
       /* temp_mat[100];  */
       *temp_mat;              /* matrix to be destructively sorted */


 /* allocate memory for temp_mat */
 temp_mat = (float *)calloc(size,sizeof(float));

 /* copy data to temporary array */
 for (i = 0; i < size; i++)
   {
    temp_mat[i] = fabs(A[i]);
   }
```

```
 quicksort(0,size-1,temp_mat);

 value = temp_mat[size-1];

 free (temp_mat);
 return(value);

}

/************************************************************************
*    Name:  quicksort.c
*    Author:  Fred W. Marcaly
*    Date:   October 16, 1990
*
*    Description:  Contains modules needed to do a quick sort.
*
***********************************************************************/

#include<afmnc.h>
#include "../execs/showtime.h"

/*-----------------------Function Declarations--------------------------*/

 void
      quicksort( ),
      swap( );
 int
      partition( ),
      find_pivot( );

/*-----------------------End of Function Declarations-------------------*/

/*=====================================================================*
* Module Name:  quicksort( )
*=====================================================================*
* Description:  Performs a quick sort on the array passed in.
*
* Input:        Indices of the first and last elements of the list to be
*               sorted and the list to be sorted.
*
* Output:       Sorted list.
*=====================================================================*/
 void quicksort(i, j, A)
 int
     i,            /* index of the first item in the list to be sorted */
     j;             /* index of the last item in the list to be sorted */
 float
     A[ ];               /* array containing list of items to be sorted */
{

 float
     pivot;                                /* value of the pivot item */
 int
     debug,
     pivotindex,                           /* index of the pivot item */
     k;              /* beginning index for group of elements >= pivot */

 pivotindex = find_pivot(i,j,A);

 if (pivotindex != -1)
   {
    pivot = A[pivotindex];
    k = partition(i,j,pivot,A);
    quicksort(i,k-1,A);
    quicksort(k,j,A);
   }
```

```
}

/*=================================================================*
* Module Name:  find_pivot()
*=================================================================*
* Description:  Returns the pivot value for the next partitioning
*               iteration of the quick sort.
*
* Input:        Indices of the first and last elements of the list to be
*               partitioned during the current iteration and the list to
*               be partitioned.
*
* Output:       Index of the item which is to be the pivot.
*=================================================================*/
 int find_pivot(i,j,A)
 int
     i,        /* index of the first item in the list to be partitioned */
     j;         /* index of the last item in the list to be partitioned */
 float
     A[];          /* array containing list of items to be partitioned */
{
 int
     k;                      /* runs right looking for another item value */
 float
     firstkey;                     /* value of the first item in the list */

 firstkey = A[i];

/* changed < to <= */
 for(k = i+1; k <= j; k++)
   {
    if (A[k] > firstkey)
      {
       return(k);
      }
    else if (A[k] < firstkey)
      {
       return(i);
      }
   }

 return(-1);

}

/*=================================================================*
* Module Name:  partition()
*=================================================================*
* Description:  Partitions the list around the pivot so that all numbers
*               greater than the pivot are on the right of the pivot and
*               all numbers less than the pivot are on the left of the
*               pivot.
*
* Input:        Indices of the first and last elements of the list to be
*               partitioned, pivot and the list to be partitioned.
*
* Output:       Partitioned list.
*=================================================================*/
 int partition(i,j,pivot,A)
 int
     i,        /* index of the first item in the list to be partitioned */
     j;         /* index of the last item in the list to be partitioned */
 float
     pivot, /* value of the item about which partitioning is to be done */
     A[];          /* array containing list of items to be partitioned */
```

```
{

 int
     L,
     R;
 float
     temp;

 L = i;
 R = j;

 while (R >= L)
   {
/*    swap(&A[L], &A[R]);       */
      temp = A[L];
      A[L] = A[R];
      A[R] = temp;

    while (A[L] < pivot)
      L++;
    while (A[R] >= pivot)
      R--;
   }

 return(L);
}

/*=====================================================================*
* Module Name:  swap()
*=====================================================================*
* Description:  Swaps the values of two variables.
*
* Input:        Pointers to the variables.
*
* Output:       Swapped values in variables.
*=====================================================================*/
 void swap(x,y)
 float
      *x, /* pointer to first variable to be swapped */
      *y; /* pointer to second variable to be swapped */

{

 float
      temp; /* temporary storage */

 temp = *x;
 *x = *y;
 *y = temp;

}
```

Vita

The author was born in Wilmington, Delaware on November 11, 1964. As a child, he was driven by some unknown urge to take apart every toy given to him to see how it worked. Although there were often parts left over when he put them back together, they still worked, at least well enough. Stuffed animals and balloons were a real problem. This preoccupation with how things worked lead him to earn a Bachelor of Science degree in Mechanical Engineering from Widener University in Chester, Pennsylvania. Along the way, he developed an avid interest in automotive racing. After finishing his Master of Science degree in Mechanical Engineering at Virginia Polytechnic Institute and State University, he plans to begin a career with General Motors. Perhaps one day, he will have the opportunity to realize his dream of racing in the Indianapolis 500.