Fitting Data of Arbitrary Dimension with B-Splines and Applications to Colour Calibration

by

Bruce Hickey

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1994

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

A fitting system for multidimensional printer gamut data was produced. The resultant B-Spline hyper-volume provides a functional definition of a printer's output capabilities. Additional system features determine the closest point on a fitted hyper-surface to a point on its exterior. This permits optimal reproduction of all colours, including those not precisely obtainable by an output device. Visualization is used to determine fit quality based on characteristics such as shape and point interpolation. The fit provides a solution to the problem of producing the best printer output of a screen image.

Acknowledgments

Many thanks are due to my supervisor, Richard Bartels, for his support, both in terms of monies and knowledge. Through countless meetings he made me feel like mine was the only project he had to be concerned with, though myriad other obligations bided for his attention. The result of these meetings was not only this thesis but also a respect and admiration of his abilities as a mentor.

My faculty readers Steve Mann and Wei-Pai Tang also are deserving of gratitude for the time spent perusing what I believed to be my final draft. This final manuscript reflects their attention to detail.

Serving as both student reader and provider of the colour theory expertise and data used in this work, Ian Bell's contributions deserve acknowledgment and thanks.

Financial support for this thesis was provided by the Natural Sciences and Engineering Research Council of Canada and the Computer Science Department of the University of Waterloo through research and teaching assistant positions respectively.

The environment within the Computer Graphics Lab cannot go unmentioned. It is the domain of people like Frank Henigman, who provided the programs to automatically generate the postscript pages for Appendix B and numerous other non-academic diversions, Steve Mann, owner of countless ice cream recipe books and the enthusiasm to try them all, and Al Vermeulen, who got me interested in the world of objects.

Having completed this work outside the CGL I'm thankful for the efforts of Fabrice Jaubert and Anne Jenson in getting drafts printed, distributed and approved.

Contents

List of Figures

Chapter 1

Introduction

A fundamental aspect of all science is the collection of observational data. Determining a compact mathematical representation that approximates such data is a frequent task in a variety of domains such as signal processing, geometric modeling and statistical analysis. In any of the these fields the fitting process may be motivated by several goals, including *parameter estimation* [19], which derives parameter values from the original data. Fitting also serves the need to remove noise from data via *smoothing* [20]. Extracting the essence of large quantities of data is accomplished via *data reduction* [22]. The last objective, determining a *functional representation*, is the goal of this work. Finding such a representation involves the approximation of an unknown function from sampling at discrete points, which are scattered throughout the unknown's domain.

Colour reproduction is a fundamental problem that has been known to graphic artists for some time. With the widespread availability of colour digital printers the problem is being faced by an increasing percentage of computer users. Currently, high quality colour reproduction is performed by a printing expert whose familiarity

with specific pigments and papers permit the selection of printer inks to reproduce select monitor colours. This is not a complete solution as even the widely used Pantone system has problems when the highly controlled environment of its special papers and dyes is expanded to include video devices. Ensuring that the colours seen on a monitor will be faithfully reproduced by a digital printer has been termed 'cross rendering' [33]. Few people are exempt from the problem, as the forward to both [2] and [17] provide anecdotal evidence of the pains suffered by even knowledgeable graphics researchers when tackling this problem.

Given that printers do not produce the exact colours requested of them, an obvious solution is to first determine which colour to ask for, to get the desired colour. An additional hurdle is presented by those colours obtainable in only one of two media. What approximation techniques [23] can be used to obtain reasonable substitutes for these unobtainable colours?

Spline mathematics represent a powerful tool [3] in the computation of formulations that are both continuous and proximate to observational data. Spline curves underlie the fitting process in all the preceding references and are used in this work to describe the characteristics of colour printers.

This work has dealt with three problems. First, how to extend existing data fitting schemes to handle data of arbitrary dimensions. Second, the application of this process to the problem of colour reproduction. The third problem was the design and implementation of tools using existing spline based classes to accelerate the completion of applications providing solutions to the first two problems. These goals were accomplished by building on existing C++ classes to allow future research to continue without the overhead of an expensive learning process. Both experienced and novice users were envisioned when implementation design was undertaken. A high level interface was designed to allow continued use of these C++

tools by researchers not concerned with their implementation details.

The work presented here unites the two problems of data fitting and colour reproduction within the framework of the Splines Project at the University of Waterloo. Empirical data was collected, representing the exact colours produced by a printer, based on controlled inputs. This data was fit using a combination of existing tools and custom built libraries. The ability to build on existing tools is further proof that the motivation for creating the Splines Project was well conceived.

Chapter 2 provides a comprehensive overview of the vocabulary, notation and mathematics used throughout the remainder of the work.

The treatment of the curve fitting process in Chapter 3 provides the opportunity to detail the approximation method in a simple context. The use of the curve fitting technique on data sets containing an arbitrary number of variables provides background for the multi-variable fitting process in the next chapter.

With all the groundwork laid, Chapter 4 describes the generalized fitting method. Optimizations to the method are also presented to permit the efficient computation of a multi-variable spline formulation.

The approximation criterion supplies a numerical estimate of the quality of the fitting spline. These distilled values are supported by the visualization of the computed formulations. This further examination provides insights about both the fit and the original data far beyond the simple numeric measure. Types of data visualization and results are presented in Chapter 5 .

Once a satisfactory functional approximation has been found for the observational data, the task of finding its inverse must be addressed. Since splines do not have a general formula describing their inverse, a minimization process must be used. Additional details on the need for this process as well as the method used

are given in Chapter 6.

Chapter 7 recounts the development of the supporting C++ classes. Examples are provided to demonstrate the usage of each individual tool. An outline of the procedure used to develop prototype applications is also given. Details of how the existing Splines libraries were leveraged to speed up the development process are given throughout this chapter.

This project has only begun to explore the usefulness of prototyping tools. Chapter 8 provides a discussion of potential future directions based on the experiences garnered from this development.

Due to the complex nature of the notation in this work a summary is presented in Appendix A.

Manual pages for the all the tools developed are in Appendix B. Other collateral work is cited as appropriate throughout this thesis.

Chapter 2

Preliminaries

It is essential to have a firm foundation on which to present the details of the work completed. Given the extensive use of multidimensional data, an overview of the structure is presented first, followed by a description of the spaces in which these data sets lie. The essential linear algebra is then laid out to allow for the presentation of the spline mathematics required to complete the entire fitting process.

The notational conventions outlined here will be used consistently throughout this work and are summarized in Appendix A.

2.1 Data Sets

This investigation focuses on the approximation of a mapping $\delta \mapsto \alpha$ by a spline function C, that is $C(\delta) \approx \alpha$. A *data set* provides the given representation of the mapping from the sampling variable, δ, to the result value, α. A data set will consist of the selected values $\delta_1, \ldots, \delta_s$ in a chosen range $low \leq \delta \leq high$, together with corresponding result values $\alpha_1, \ldots, \alpha_s$.

Example 2.1 *A monochrome printer is to demonstrate its abilities by producing five swatches with various intensities of gray, measured as percentages of black. The shades of gray to be printed are measured by generating the input samples*

$$\{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5\} = \{20\%, 40\%, 60\%, 80\%, 100\%\}.$$

Printers, being fallible physical devices, will not respond exactly to their input. The grey levels actually printed are measured to be

$$\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\} = \{19\%, 37\%, 62\%, 80\%, 98\%\}.$$

Together these samples and results form the data set

$$\{\{20\%, 19\%\}, \{40\%, 37\%\}, \{60\%, 62\%\}, \{80\%, 80\%\}, \{100\%, 98\%\}\},$$

where $s = 5$, $low = 20\%$, $high = 100\%$.

To approximate the input/output mapping of this printer we might choose a spline that is, for example, composed of the four basis functions $B_1(\delta)$, $B_2(\delta)$, $B_3(\delta)$, $B_4(\delta)$*:*

$$C(\delta) = \sum_{j=1}^{4} \nu_j B_j(\delta).$$

In the approximation of the mapping $\delta \mapsto \alpha$ the values of $\nu_1, \nu_2, \nu_3, \nu_4$ must be selected so that

$$C(\delta_i) = \sum_{j=1}^{4} \nu_j B_j(\delta_i) \approx \alpha_i \qquad (i = 1, \ldots, 5).$$

This is a classical curve fitting problem, whose solution is computed by solving the linear system

$$\begin{bmatrix} B_1(\delta_1) & B_2(\delta_1) & B_3(\delta_1) & B_4(\delta_1) \\ B_1(\delta_2) & B_2(\delta_2) & B_3(\delta_2) & B_4(\delta_2) \\ B_1(\delta_3) & B_2(\delta_3) & B_3(\delta_3) & B_4(\delta_3) \\ B_1(\delta_4) & B_2(\delta_4) & B_3(\delta_4) & B_4(\delta_4) \\ B_1(\delta_5) & B_2(\delta_5) & B_3(\delta_5) & B_4(\delta_5) \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \\ \nu_4 \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix}$$

in a least squares sense.

To handle colour reproduction in devices such as a printer or monitor, a data set must be *multidimensional.* Each dimension in the data set reflects one of several input values, such as a percentage of certain dyes or phosphor excitation. These inputs will produce an output colour that may require specification in terms of several spectral components.

Example 2.2 *A printer uses three types of toner: $\delta^1 \equiv$ cyan, $\delta^2 \equiv$ magenta, $\delta^3 \equiv$ yellow. Input commands are given in terms of a percentage of maximum concentration, so that each input might be sampled at selected values. In this example the values of δ^1 are $low^1 = 0\% \leq \delta^1 \leq high^1 = 90\%$*

$$\{\delta_1^1, \delta_2^1, \delta_3^1, \delta_4^1, \delta_5^1\} = \{0\%, 20\%, 40\%, 80\%, 90\%\}$$

δ^2 are $low^2 = 20\% \leq \delta^2 \leq high^2 = 100\%$

$$\{\delta_1^2, \delta_2^2, \delta_3^2, \delta_4^2\} = \{20\%, 40\%, 80\%, 100\%\}$$

and finally δ^3 *are* $low^3 = 10\% \leq \delta^3 \leq high^3 = 95\%$

$$\{\delta_1^3, \delta_2^3, \delta_3^3, \delta_4^3 \delta_5^3, \delta_6^3, \} = \{10\%, 20\%, 30\%, 60\%, 75\%, 95\%\}.$$

The colour outputs of the device are typically measured as linear combinations, where the $*$ *operator multiplies each of the vector elements by the given co-efficient, of some primary spectra, consisting of a vector of discrete wavelengths, for example:*

$$\alpha^1 * spectrum^1 + \alpha^2 * spectrum^2 + \alpha^3 * spectrum^3 + \alpha^4 * spectrum^4.$$

This means that for each selected input value

$$(\delta^1, \delta^2, \delta^3)$$

a measured spectral decomposition

$$(\alpha^1, \alpha^2, \alpha^3, \alpha^4)$$

would result, yielding the data set[1]

$$\left\{ \begin{array}{c} \{(\delta_1^1, \delta_1^2, \delta_1^3); (\alpha_{1,1,1}^1, \alpha_{1,1,1}^2, \alpha_{1,1,1}^3, \alpha_{1,1,1}^4)\} \\ \vdots \\ \{(\delta_5^1, \delta_4^2, \delta_6^3); (\alpha_{5,4,6}^1, \alpha_{5,4,6}^2, \alpha_{5,4,6}^3, \alpha_{5,4,6}^4)\} \end{array} \right\}.$$

Now, the approximating spline becomes more complicated. If a tensor product spline is used, as in this work, a number of basic splines for each input variable $\delta^1, \delta^2, \delta^3$ *are individually chosen. As an example, for* δ^1*:*

$$B_1^{(1)}(\delta^1),\ B_2^{(1)}(\delta^1),\ B_3^{(1)}(\delta^1),\ B_4^{(1)}(\delta^1)$$

for δ^2*:*

$$B_1^{(2)}(\delta^2),\ B_2^{(2)}(\delta^2)$$

[1] The subscripts on the α's reflect the fact that each α depends on each of the three δ values.

and for δ^3:

$$B_1^{(3)}(\delta^3),\ B_2^{(3)}(\delta^3),\ B_3^{(3)}(\delta^3).$$

The spline itself takes the form

$$C(\delta^1,\delta^2,\delta^3) = \sum_{j_1=1}^{4}\sum_{j_2=1}^{2}\sum_{j_3=1}^{3} \nu_{j_1,j_2,j_3} B_{j_3}(\delta^3)B_{j_2}(\delta^2)B_{j_1}(\delta^1)$$

with each of the ν's having the same number of components as there are α's

$$\nu_{j_1,j_2,j_3} = (\nu^1_{j_1,j_2,j_3}, \nu^2_{j_1,j_2,j_3}, \nu^3_{j_1,j_2,j_3}, \nu^4_{j_1,j_2,j_3}).$$

This allows the precise restatement of the multidimensional fitting problem as the solution to

$$C(\delta^1_{i_1}, \delta^2_{i_2}, \delta^3_{i_3}) \approx (\alpha^1_{i_1,i_2,i_3}, \alpha^2_{i_1,i_2,i_3}, \alpha^3_{i_1,i_2,i_3}, \alpha^4_{i_1,i_2,i_3}).$$

In a multidimensional environment it is easy for the various constituents (the number of indices, the range of these indices, the number of input components and the number of output components) to blend together and overwhelm the reader. This leads to the definitions and notational conventions that follow.

The attributes measured by the result values, $\alpha^1, \ldots \alpha^t$ are termed the *result components*, with their number denoted by t. This number of result components is independent of the *sampling dimension*, represented by n, which is the dimension of the space spanned by the *sampling variables* $\delta^1, \ldots, \delta^n$. A sampling set, δ, has a total of s elements. In the multidimensional case, where several sampling sets exist, the i-th set, δ^i, is considered to have s_i elements.

Example 2.1 outlined a situation where $t = 1$, $n = 1$ and $s = 5$. Expanding this example, by measuring not only the intensities, but also the thicknesses of toner on the page, would demonstrate the independence of sampling and the number of

result components. This new measurement would give $t = 2$ result components, while maintaining $n = 1$.

Example 2.2 demonstrates the case where $n > 1$. Here, $t = 4$, $n = 3$, while $s_1 = 5, s_2 = 4$ and $s_3 = 6$. To represent such data the concept of a *multidimensional data set* is introduced.

Definition 2.1 *A* **multidimensional data set**, D^n, *consists of the rectangular lattice with* $(n+t)$*-tuples as entries. The structure of the lattice is defined by Cartesian product of the n sampling sets,* $\delta^1 \times \delta^2 \times \ldots \times \delta^n$, *implying* D^n *is an element of* $\mathbb{R}^{s_1 \times s_2 \times \ldots \times s_n \times (n+t)}$. *The entries at each lattice point contain both a sampling point,* δ, *and the result values,* α. *Therefore, a typical entry would have the form* $(\delta^1, \ldots, \delta^n; \alpha^1, \ldots, \alpha^t)$.

Definition 2.2 *A data set,* D^n, *based on n sampling sets has a* **data dimension** *of n.*

To simplify future notation the symbol $\mathbb{L}_{n+t}$ will be used to represent the individual lattice elements, which are $(n+t)$-tuples. When the tuple size is implied the subscript is omitted. The benefit of this concept is demonstrated in the condensed representation of data sets, i.e $D^n \in \mathbb{L}^{s_1 \times s_2 \times \ldots \times s_n}$.

As a tensor product, D^n has several properties:

- Any entry of D^n, or component thereof, in the lattice can be addressed by the indices that combine to form it.
- D^n has $\prod_{j=1}^{n} s_j$ elements.
- For any data dimension, i, there are $(\prod_{j=1}^{n} s_j)/s_i$ 1-variable vectors of elements from $\mathbb{L}$ in the set D^n.

Example 2.3 *The entry* $(\delta^1_{i_1}, \delta^2_{i_2}, \delta^3_{i_3}; \alpha^1_{i_1,i_2,i_3}, \alpha^2_{i_1,i_2,i_3}, \alpha^3_{i_1,i_2,i_3}, \alpha^4_{i_1,i_2,i_3})$ *in the data set from Example 2.2 would have the indices* i_1, i_2, i_3 *and the components* δ^1, δ^2, δ^3, α^1, α^2, α^3, α^4.

These properties are used to determine the number of *slices* contained within a lattice.

Definition 2.3 *A* **slice**, θ_i, *is a vector of* $(n+t)$*-tuples extracted from a multidimensional data set by holding all indices constant except the one given by* i.

Example 2.4 *Given the data set* $D \in \mathbb{L}_7^{5\times4\times6}$ *from Example 2.2*

$$\theta_2 = \{d_{2,i_2,1}\}_{i_2=1}^{4}$$

specifies a slice of data in the second data dimension, whose elements are

$$\begin{aligned}
d_{2,\mathbf{1},1} &= \{\delta^1_2, \delta^2_{\mathbf{1}}, \delta^3_1; \alpha^1_{2,\mathbf{1},1}, \alpha^2_{2,\mathbf{1},1}, \alpha^3_{2,\mathbf{1},1}, \alpha^4_{2,\mathbf{1},1}\} \\
d_{2,\mathbf{2},1} &= \{\delta^1_2, \delta^2_{\mathbf{2}}, \delta^3_1; \alpha^1_{2,\mathbf{2},1}, \alpha^2_{2,\mathbf{2},1}, \alpha^3_{2,\mathbf{2},1}, \alpha^4_{2,\mathbf{2},1}\} \\
d_{2,\mathbf{3},1} &= \{\delta^1_2, \delta^2_{\mathbf{3}}, \delta^3_1; \alpha^1_{2,\mathbf{3},1}, \alpha^2_{2,\mathbf{3},1}, \alpha^3_{2,\mathbf{3},1}, \alpha^4_{2,\mathbf{3},1}\} \\
d_{2,\mathbf{4},1} &= \{\delta^1_2, \delta^2_{\mathbf{4}}, \delta^3_1; \alpha^1_{2,\mathbf{4},1}, \alpha^2_{2,\mathbf{4},1}, \alpha^3_{2,\mathbf{4},1}, \alpha^4_{2,\mathbf{4},1}\}
\end{aligned}$$

Since θ_2 *is being examined, the second index is in bold type to highlight the sequence.*

Typically, we are interested in the set of all slices available in a given data dimension, defined by:

Definition 2.4 *A* **slice set**, Θ_i, *is the set of all the slices of a multidimensional data set, in the* i*-th dimension.*

In the following example note that 'dimension' implies the data dimension:

Example 2.5 *The data set $D \in \mathbb{I}\!\mathbb{L}^{5\times4\times6}$ can be visualized as a rectangular volume, or a set of matrices stacked one upon another. In the first dimension the set of slices would be five matrices from $\mathbb{I}\!\mathbb{L}^{4\times6}$. Those matrices can be visualized as slicing the volume from front to back. In the second dimension, slicing from left to right would provide four matrices from $\mathbb{I}\!\mathbb{L}^{5\times6}$. Finally, in the third dimension, which slices from top to bottom, there would be six matrices from $\mathbb{I}\!\mathbb{L}^{5\times4}$.*

The slice set notation permits the compact representation of arbitrary views into data sets.

Example 2.6 *Given the data set $D \in \mathbb{I}\!\mathbb{L}^{5\times4\times6}$,*

$$\Theta_2 = \{\theta_2\}_{i_1=1,i_3=1}^{5,6} = \{\{d_{i_1,i_2,i_3}\}_{i_2=1}^{4}\}_{i_1=1,i_3=1}^{5,6}$$

specifies the set of slices in the second data dimension.

Each slice set consists of all the elements in the data set. It is simply their ordering that is being manipulated. As such, and given space restrictions, the $s_1 s_2 s_3 = 5 * 4 * 6 = 120$ elements of the slice set in Example 2.6 are not expanded here.

2.2 Colour Spaces

The data used during this project came from various sources, including collaborative research at the Computer Graphics Lab at the University of Waterloo and industrial labs, all pursuing similar goals in colour reproduction. Understanding these data sets involves some knowledge of the colour spaces in which the measurements were taken.

RGB

This **R**ed, **G**reen, **B**lue model is used by colour monitors and is an additive model based in a Cartesian system. It is shown here as in [15]

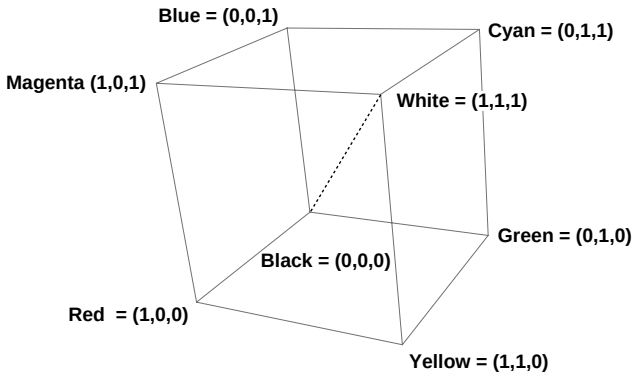

Figure 2.1: The RGB colour space.

CMY

Cyan, **M**agenta, **Y**ellow is a subtractive colour model used by hardcopy devices. Its components are defined to be

$$[C, M, Y] = [1, 1, 1] - [R, G, B],$$

meaning that they are found on complementary corners of the RGB cube, as shown in Figure 2.1.

CMYK

The addition of black, represented by K, to the CMY model accommodates the physical limitations of most colour printers. Ideally, the application of equal levels of CMY inks at a specified pixel will produce black at that location. True black is seldom the result of this process and the new model is able to avoid this situation. Given a point in CMY space its counterpart in CMYK space is defined by

$$\begin{aligned} K &= min(C, M, Y) \\ C &= C - K \\ M &= M - K \\ Y &= Y - K \end{aligned} \tag{2.1}$$

CIELAB

In an attempt to define a model in which the distance between two colours, measured by the Euclidean distance, would be proportional to the relative perceptual

distance of those colours the CIELAB space was developed. As noted by [31] L^* is a measure of lightness and a^* and b^*, which do not correspond to any known properties of visual perception, can be considered to indicate changes in red/green and green/blue balance respectively.

CIELAB describes the entire space of visible light and therefore contains typical monitor and printer gamuts within its boundaries. The following diagram, adapted from [27], demonstrates how some colours cannot be obtained by both devices.

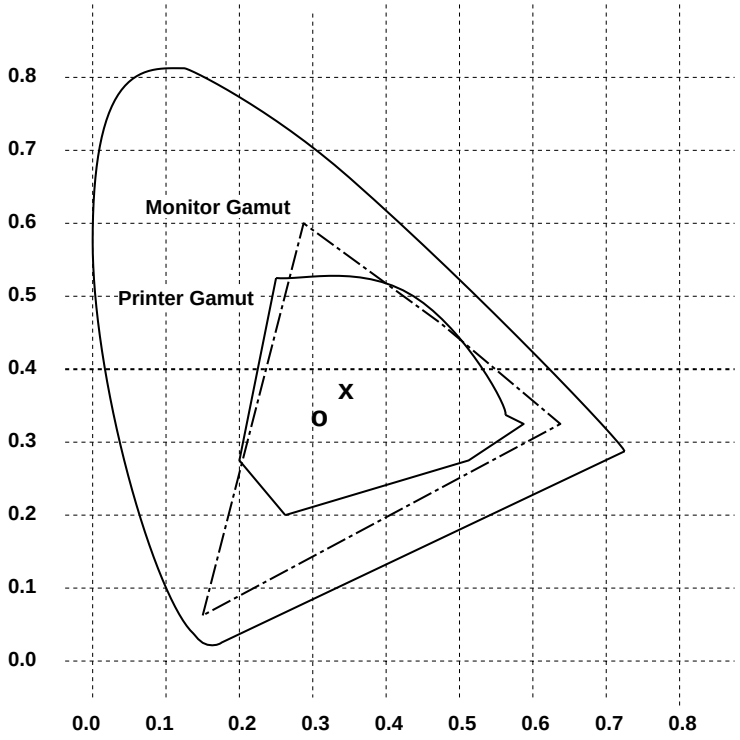

Figure 2.2: Printer and Monitor Gamuts Within CIE Chromaticity Space, with X and O representing the white point for the printer and monitor gamuts respectively.

Alpha-bases

The Alpha basis functions are computed from Singular Value Decompositions, or SVDs, of spectroradiometric data. These orthogonal bases span the entire reflectance space, providing means to describe any colour as a linear combination, as mentioned in Example 2.2. The exact shape of these basis spectra are beyond the scope of this work; details are presented in [28]. The use of $\alpha^1, \ldots, \alpha^t$, as output components serves to remind the reader that these are coefficients to the alpha bases.

Additional colour spaces such as YIQ, used as an NTSC broadcast standard, and HSV, which uses perceptual properties of colour, do exist. The reader is referred to [31] [21] for complete treatments of colour spaces for which data sets were not available.

2.3 Linear Algebra for Least-Squares Approximation

As will be shown later, the task of fitting a spline to a set of data collapses to the problem of solving a linear system of the form

$$CV = A.$$

The solution, $V = \{\nu_i\}_{i=1}^{m}$, to such a system is computed from the known data vector, $A = \{\alpha_i\}_{i=1}^{s}$, and the matrix, $C = \{C_{i,j}\}_{i=1,j=1}^{s,m}$. No restriction is placed on A, while C must have linear independence amongst its columns. The variable s continues to represent the number of entries in the sampling set, while m is used only as a placeholder. The variable m will be associated with a characteristic of the fitting problem in Section 2.4.

If an exact solution cannot be found, the goal is to compute values for V that minimize the Euclidean norm of $CV-A$. Complete treatments of this topic can be found as a discussion of numerical techniques in [8] and more directly as a spline fitting problem in [32, 9].

The problems defined by such linear systems can be separated into three types:

- $s = m$, defines a system with a unique solution, satisfying (2.2).
- $s > m$, yields an overdetermined system whose unique solution provides a residual of minimal Euclidean norm.
- $s < m$, describes a problem with a subspace of solutions in $V \in \mathbb{R}^m$.

Only the first two cases are of interest. As will be shown in Section 2.3.2, the last case cannot underlie a curve fitting problem.

2.3.1 Solving the Linear System

The ensuing discussion is modeled after the work presented by Srećković [32]. At this time, assume that C is in triangular form. Finding the solution V to those systems where $s = m$, and whose components are

$$\begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,m} \\ 0 & C_{2,2} & \cdots & C_{2,m} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & C_{m,m} \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \\ \vdots \\ \nu_m \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_s \end{bmatrix}, \tag{2.2}$$

involves the application of two substitutions:

$$\nu_m = \frac{\alpha_s}{C_{m,m}}, \tag{2.3}$$

and

$$\nu_i = \frac{1}{C_{i,i}} \left(\alpha_i - \sum_{j=i+1}^{s} C_{i,j} \nu_j \right), \tag{2.4}$$

for $i = s-1, \ldots, 1$. There is no possibility for division by zero with any of the elements $\{C_{i,i}\}_{i=1}^{m}$, as the linear independence of the columns ensures that all the diagonal elements are non-zero.

When $s > m$ the system will have the form:

$$\begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,m} \\ 0 & C_{2,2} & \cdots & C_{2,m} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & C_{m,m} \\ 0 & \cdots & 0 & 0 \\ \vdots & \cdots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \\ \vdots \\ \nu_m \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \\ \alpha_{m+1} \\ \vdots \\ \alpha_s \end{bmatrix}.$$

The solution vector V is again found through the substitutions of (2.3) and (2.4). Diagonal elements are, as before, non-zero by linear independence.

Those elements of A directly influencing the solution, V, are now extracted to form

$$A_u = \begin{bmatrix} \alpha_1 & \cdots & \alpha_m & 0 & \cdots & 0 \end{bmatrix}^T.$$

The data contained in the complement to A_u

$$A_l = \begin{bmatrix} 0 & \cdots & 0 & \alpha_{m+1} & \cdots & \alpha_s \end{bmatrix}^T$$

is not used during the solution process. However, this does not imply that A_l is without meaning. The interpretation of these vectors is presented in the following section.

2.3.2 Determining a Least-Squares Solution

Focusing on the least-squares solution to the problem $CV = A$ implies a goal of minimizing

$$\|CV - A\|_2^2.$$

The vector A can be expressed as a vector sum,

$$\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \\ \alpha_{m+1} \\ \vdots \\ \alpha_s \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \alpha_{m+1} \\ \vdots \\ \alpha_s \end{bmatrix}, \tag{2.5}$$

in which case

$$\|CV - A\|_2^2 = \|CV - A_u - A_l\|_2^2.$$

Since the inner products $A_l^T A_u$ and $V^T C^T A_l$ are both zero, Equation (2.5) can be simplified.

$$\begin{aligned} \|CV - A_u - A_l\|_2^2 &= (CV - A_u - A_l)^T (CV - A_u - A_l) \\ &= V^T C^T CV - V^T C^T A_u - V^T C^T A_l + A_u^T A_u + A_u^T A_l + A_l^T A_l \\ &= (CV - A_u)^T (CV - A_u) + A_l^T A_l \\ &= \|CV - A_u\|_2^2 - \|A_l\|_2^2 \end{aligned} \tag{2.6}$$

With an exact solution to the system $CV = A_u$, the first component of Equation (2.6) is eliminated by the fact that $CV - A_u = 0$. The least-squares residual can now be found within the entries of the hereto unused subvector, A_l.

$$\|CV - A\|_2^2 = \|A_l\|_2^2$$

2.3.3 Matrix Factorization

To this point, only those systems having C in triangular form have been considered. In practice, however, this ideal structure rarely presents itself. An orthogonal factorization process, such as QR-factorization, may be used to provide the desired triangular structure within C.

Definition 2.5 *The* **QR-factorization** *of a matrix $C \in \mathbb{R}^{s \times m}$ yields an orthogonal matrix*[1] *$Q_C \in \mathbb{R}^{s \times s}$ and an upper triangular*[2] *matrix $R_C \in \mathbb{R}^{s \times m}$ such that $Q_C R_C = C$*

[1] The matrix Q_C can be computed numerically provided that the columns of C provide *sufficient linear independence* in relation to the precision of calculation, see [18].

[2] While normally used to describe square matrices, *upper triangular* is used here for rectangular matrices having $C_{i,j} = 0$ whenever $i > j$.

The factorization is accomplished through a series of orthogonal decomposition matrices $\{H_i\}_{i=1}^{m}$, whose form will be detailed in Section 3.5.

Definition 2.6 *A QR-factorization of a matrix $C \in \mathbb{R}^{s\times m}$ can be expressed in* **product form** *as:*

$$H_m H_{m-1} \cdots H_1 C = R_C.$$

Since each H_i is orthogonal, this expression can be rewritten as:

$$H_1^T \cdots H_{m-1}^T H_m^T R_C = C.$$

The reader is referred to [18] for proof that such a decomposition can always be computed and that the following theorem is true.

Theorem 2.1 *If H is an orthogonal transformation then*

$$\|H(CV - A)\|_2^2 = \|HA_l\|_2^2.$$

With a QR-factorization it is possible to transform the general system $CV = A$ into one which has the desired upper triangular matrix form. This transformation is applied as follows:

$$\begin{aligned} CV &= A \\ Q_C R_C V &= A \\ Q_C^T Q_C R_C V &= Q_C^T A \\ R_C V &= Q_C^T A. \end{aligned} \tag{2.7}$$

2.4 Splines

For reasons detailed in Section 3.1, the fitting process uses the nonuniform B-spline formulation to approximate the sampled colour data. A brief overview of the fundamental properties and attributes of these splines is presented here. Exhaustive treatments of spline theory and proofs can be found in [4, 14]

Splines are piecewise polynomials that may be built as linear combinations of basis splines and coefficients taken from $\mathbb{R}$. The B-splines are basis splines introduced by Schoenberg [9].

Definition 2.7 *A* **knot sequence** *is a non-decreasing set of numbers.*

The symbol μ is used to denote elements of the knot vector. The knot vector helps determine the shape and define the continuity of the spline by providing a sequence of *intervals* and a set of *multiplicities.*

Definition 2.8 *The* **multiplicity** *of a knot μ is the number of occurrences of μ in the knot sequence.*

The multiplicity of a specific knot will be denoted by q. For any point within the knot sequence the notion of its *interval* is defined as follows.

Definition 2.9 *The* **interval** *of a point u is a left-open range of values $(\mu_l, \mu_{l+1}]$ that contains u, $\mu_l < u \leq \mu_{l+1}$.*

Definition 2.10 *The* **interval index** *of a point u is the l satisfying Definition 2.9.*

Definition 2.11 *The* **order** *of a B-spline basis is defined to be one plus the degree of the polynomials that comprise the basis functions.*

The order of a B-spline is referred to as r. Together, the knot sequence and the order define a set of basis functions $\{B_i^r\}_{i=1}^m$. This value for m is the same value given in Section 2.3; the value and meaning of m will be described later. Returning to the definition of the basis functions:

Definition 2.12 *The i-th* **B-spline** *of order r over a knot sequence $\{\mu_i\}_{i=0}^{m+r}$ is defined as*

$$B_i^1 \stackrel{\text{def}}{=} \begin{cases} 1, & \mu_i < u \leq \mu_{i+1} \\ 0, & \textit{otherwise} \end{cases} .$$

$$B_i^r(u) \stackrel{\text{def}}{=} \frac{u - \mu_i}{\mu_{i+r-1} - \mu_i} B_i^{r-1}(u) + \frac{\mu_{i+r} - u}{\mu_{i+r} - \mu_{i+1}} B_{i+1}^{r-1}(u),$$

for $r = 2, 3, \ldots, m$ and $i = 1, 2, \ldots, m$,

with the provision that no knot has multiplicity greater than r, that

$$\frac{u - \mu_i}{\mu_{i+r-1} - \mu_i}$$

is interpreted as zero whenever $\mu_{i+r-1} - \mu_i = 0$, and that

$$\frac{\mu_{i+r} - u}{\mu_{i+r} - \mu_{i+1}}$$

is interpreted as zero whenever $\mu_{i+r} - \mu_{i+1} = 0$.

The undefined term m in Definition 2.7 and Equation 2.2 is actually the number of B-splines that are produced by Definition 2.12. Each of the B-splines are linearly independent and they span a space that has a *spline space dimension* of m.

Definition 2.13 *The* **spline space dimension** *of a set of B-splines, defined by Definition 2.12, is the dimension of the space spanned by those basis functions.*

It follows that the spline space dimension of the B-spline is the number of knots in the basis, minus the order of the basis.

It can be shown from the definition that $B_i^r(u)$ is zero if $u < \mu_i$ or $u > \mu_{i+r}$. This means that within the first $r-1$ knot intervals there are fewer than r non-zero basis functions, leading to the notion of an interval being *fully covered.*

Definition 2.14 *An interval* $(\mu_l, \mu_{l+1}]$ *is said to be* **fully covered** *if each of its basis functions* $\{B_i^r\}_{i=l}^{l-r+1}$ *are non-zero. The* r *B-splines, when restricted to this interval, are all linearly independent polynomials of order* r*.*

Similarly, the last $r-1$ intervals are not fully covered either.

Definition 2.15 *The* **control vertices** $V = \{\nu_i\}_{i=1}^m$*, where* $\nu_i \in \mathbb{L}_t$[2]*, are the coefficients used in linear combination with the B-splines to form a B-spline curve.*

The number of control vertices is given by the spline space dimension, m, while each control vertex is a point in the space $\mathbb{L}_t$.

Definition 2.16 *A* **B-spline curve** $C(u)$ *is a linear combination of the B-spline functions* $B_i^r(u)$ *and control vertices* $V = \{\nu_i\}_{i=1}^m$ *defined by:*

$$C(u) = \sum_{i=1}^{m} \nu_i B_i^r(u)$$

[2]Recall that in the single variable case $\mathbb{L}_t \equiv \mathbb{R}^t$

The space of the control vertices determines the space in which the curve exists. For example, a planar curve would have $\nu_i \in \mathbb{R}_2$, while a curve in 3-space would have control vertices in $\mathbb{R}_3$.

A curve can also be referred to as a *single-variable* spline, as it has only one parameter. Later, *multi-variable* tensor product splines will be described; however, it is worthwhile to note here that the number of variables is independent of all other spline attributes. The number of variables used in a specific problem is determined by the number of inputs that produce the data set being approximated.

Chapter 3

Spline Fitting in One Variable

The advent of splines provided new formulations for the problem of determining functional approximations to empirical data [6]. This chapter begins with a justification of the types of splines chosen to underlie the fitting process. A review of the details of the spline fitting algorithm follows. Finally, special characteristics exhibited by the chosen splines and colour data will be examined to optimize an existing least squares implementation.

3.1 Motivation

While the basis of this work is the fitting of multidimensional data, an explanation of the simplest case will make later results more tractable. This permits the introduction of notation that will necessarily be quite complex with splines of an arbitrary number of variables.

Hand in hand with a linear increase in the dimension of data domain, given by the sampling points, comes an exponential increase in the size of data sets.

Optimizations of the basic curve fitting algorithm were needed in the face of this increased computational workload. Close examination of the linear systems that need to be solved provides some simple, but effective, ways of streamlining the algorithm.

Before examining the details of the least squares fitting process, a discussion of the type of curve used to perform the fit is required. Three criteria were established for the type of spline used to fit the printer data: it should provide useful approximations and it should exhibit smoothness, in the form of continuous second derivatives and local control. To allow for quick implementation, only formulations currently implemented by the Splines project were considered. These already had tested evaluation and manipulation tools. Finally, as shown in Chapter 6, a spline that can be refined was required.

The derivative discontinuities that occur in Bézier splines, unless the appropriate restrictions are enforced, would not provide the desired smoothness. Rational splines were inappropriate for several reasons. The complexity of fitting algorithms using rationals was much higher, while the best algorithm is still being debated. More importantly, it is not apparent that the increased control provided by rationals could be effectively utilized with the given data. Uniform B-splines were implemented but would suffer from a dramatic increase in the size of the control mesh if refinement was required. The nonuniform B-spline, or NUB-spline, satisfied all of the stated criteria and was selected as the curve to underlie the fitting process.

The actual approximation process is comprised of two steps. First, the basic parameters of a fitting curve had to be set. Then the linear system generated from the chosen parameters and given data sets had to be solved, using least squares methods. A more in depth discussion of this entire process requires the introduction of some mathematical concepts.

3.2 Notation

Eliminating all but one candidate for the job of fitting the data provides the following definition:

Definition 3.1 *A* **fitting spline** $C(\delta)$ *is the NUB-spline of order* r *and dimension* m *given by*

$$C(\delta) = \sum_{i=1}^{m} \nu_i B_i^r(\delta)$$

The goal is to determine the values for the control structure V that provides a curve to best fit the data. This is subject to an *approximation criterion*, which is computed via a *residual function.*

Definition 3.2 *A* **residual function** $res(C, D)$ *computes a numeric result that describes the difference between the known result components* $\{\alpha_i\}_{i=1}^{s}$ *of the data set* D *and a fitting spline* $C(\delta)$.

This numeric measure of success may use all the information present in its arguments at once, or it may, as in this work, compute a value based on the difference between the individual data points and the curve values at a predetermined set of *evaluation points.* In either case, one evaluation point is required for each of the result values. Parametric curve fitters compute the evaluation points based on the data set while traditional curve fitting problems, such as those used in this work, simply use the sampling points provided in the data sets.

Despite having chosen the evaluation points to be exactly the sampling points, there is still a need to distinguish between them. Not all fitters share the same set for both purposes. To avoid confusion, future discussion will refer to the shared

set of points in the context in which they are used, be it as sampling or evaluation points

It is now possible to provide a more precise definition of the residual. This function incorporates the fitting curve, evaluated at each of the evaluation points, and the data points themselves. The approximation criterion used in this work is based on the Euclidean distance, or l_2 norm. It seeks to minimize the sum of distances between the data points and the fitting curve values at each of the evaluation points.

Definition 3.3 *The* **Least Squares Residual Norm** *of a fitting curve $C(\delta)$ and a set of data points $D = \{\delta_i; \alpha_i\}_{i=1}^{s}$ where $\alpha_i = (\alpha_i^1, \ldots, \alpha_i^t)$ is defined as*

$$res(C, D) = \sum_{i=1}^{s} \|C(\delta_i) - \alpha_i\| \tag{3.1}$$

The final requirement is a formal definition of the notion of a curve fit that minimizes the residual function in (3.1).

Definition 3.4 *A curve fit seeks to find control vertices $\{\nu_i\}_{i=1}^{m}$ that will provide a minimum value for*

$$\sum_{i=1}^{s} \|C(\delta_i) - \alpha_i\|$$

where $C(\delta_i)$ is the spline curve of order r, spline space dimension[3] m, and knot sequence $\{\mu_i\}_{i=0}^{m+r-1}$, and where $\{\alpha_i\}_{i=1}^{s}$ are the result components being approximated.

[3]This is the dimension of the spline, not to be confused with the dimension of the space in which the spline lives; the latter is dictated by the control vertices.

3.3 The Underlying Curve

The exact shape of the *fitting curve* is not known until the linear system is solved. Before setting up such a system, the basic attributes of the spline need to be determined.

To begin, the order of the fitting spline must be chosen. Any positive value is possible, as will be shown in Chapter 7, but is subject to two restrictions. The order is bounded by the number of data points, since the linear system it helps to define must be overdetermined. The second consideration is the evaluation of the computed fitting spline. Higher orders are significantly harder to evaluate; orders higher than five are usually avoided.

Figure 3.1 demonstrates the approximation of a data set by a curve with order four. In raising the order of the spline to five a better fit is achieved, Figure 3.2.

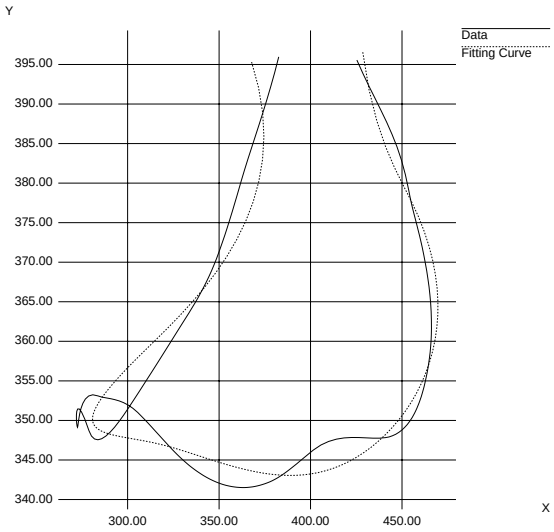

Figure 3.1: Uniform knot spacing, with an order four spline.

The second component is the knot vector. The placement of knots over the

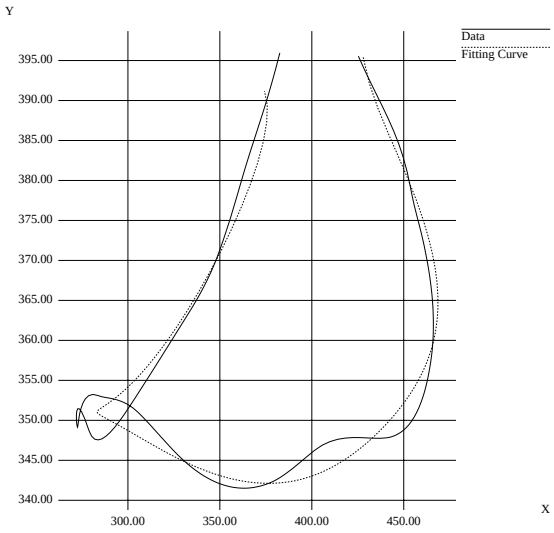

Figure 3.2: Uniform knot spacing, with an order five spline.

domain of the fitting spline determines the amount of control the curve will have in each of the intervals that comprise the curve. There are two distinct regions of interest for these values, those that control the behavior of the curve at the ends of the valid interval and those that exert control between those endpoints. Returning to a curve of order four, the the knots are arranged to force interpolation at the end points, Figure 3.3.

The increase in order is only one means of achieving a better approximation. By adding more knots the curve is better able to follow the shape of the data.

The number of segments must be carefully chosen to not be excessively influenced by noise in the data. For example, the tiny leftmost loop in data sets used in this section may only be noise in the data. Introducing too many segments would allow the fitting curve to mimic that loop. This is seen in Figure 3.4 which shows a curve with two more segments than the curve in Figure 3.2.

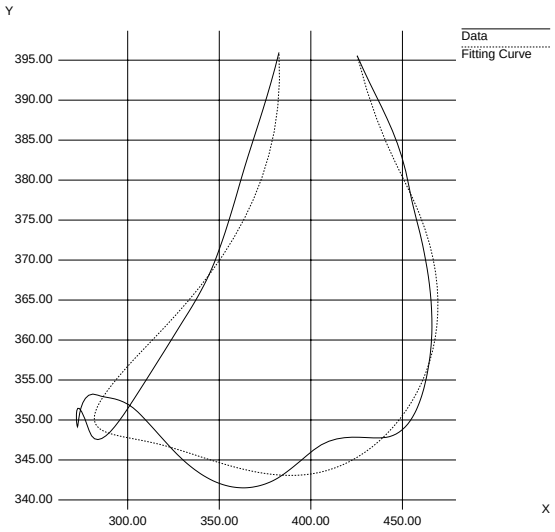

Figure 3.3: End knots having order multiplicity, with order four spline.

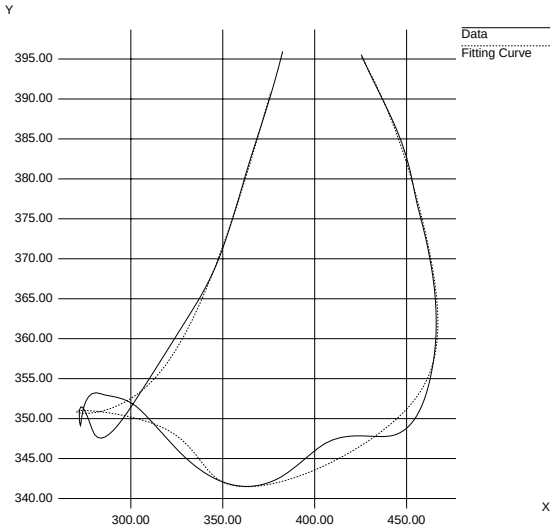

Figure 3.4: The effect of adding extra segments to an order four spline.

3.4 Solving the Linear System

The approach used to compute a fitting curve involves finding the control structure of a spline curve by solving the linear system $CV = A$. Section 2.3 outlined the need to have an overdetermined linear system. An examination of what that implies in the realm of fitting with B-splines follows.

To be overdetermined, the vector of data points, $\{\alpha_i\}_{i=1}^{s}$, must contain more elements than the solution vector $\{\nu_i\}_{i=1}^{m}$. That is to say, there must be more data points available than control points desired. Obtaining the first fully defined interval requires r control points. Additional intervals are introduced with each subsequent control point. There is no easy method to determine the appropriate number of intervals for the fitting spline. Allowing too few intervals results in poor fitting accuracy, as given by the residual function. Conversely, too many intervals will see the fitting spline affected by noise in the data. The optimal number of intervals, in general, cannot be given, though at this point the subjective restriction of having only one or two evaluation points per interval is proposed. Specific case results are presented later.

Ensuring that the linear system is overdetermined becomes a matter of satisfying

$$s > m.$$

This provides $m - r + 1$ intervals. The constraint of having the number of intervals as a whole number yields $m - r + 1 > 0$; thus

$$m \geq r.$$

Example 3.1 *A data set with s=6 values has been collected. This data is to be fit with a cubic B-spline, r=4, giving* $6 > m \geq 4$. *Since m can be either 4 or 5, the fitting spline can have either 1 or 2 intervals.*

Care must also be taken to ensure that the columns of C are linearly independent. This independence can be compromised when the evaluation points are located too near one another, resulting in basis values that are practically equal, or it can result in data that is too widely scattered, resulting in basis functions that are not evaluated at any position for which they have a nonzero value [18, 9].

Taking a closer look at the components of the system returns to the basic equation, $CV = A$. The matrix $C \in \mathbb{R}^{s \times m}$ contains the spline basis function evaluations at each of the parameter points. Each of the s rows, C_i, contain the values of the m basis functions evaluated at the point u_i. Additionally, every basis function is evaluated at least once, meaning that for every interval $(\mu_j, \mu_{j+r}]$, where $j \geq r-1$ (see Definition 2.14), there is always an evaluation point u such that $\mu_j < u \leq \mu_{j+r}$. Thus, one row of the matrix is given by

$$C_i = [B_1(u_i), B_2(u_i), \cdots, B_m(u_i)]$$

yielding the complete matrix

$$C = \begin{bmatrix} B_1(u_1) & B_2(u_1) & \dots & B_m(u_1) \\ B_1(u_2) & B_2(u_2) & \dots & B_m(u_2) \\ \vdots & \vdots & \ddots & \vdots \\ B_1(u_s) & B_2(u_s) & \dots & B_m(u_s) \end{bmatrix}.$$

The vector $A \in \mathbb{R}^s$ are the result components

$$A = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_s \end{bmatrix}.$$

The solution vector, $V \in \mathbb{R}^m$, is the control net of the fitting curve

$$V = \begin{bmatrix} \nu_1 \\ \nu_2 \\ \vdots \\ \nu_m \end{bmatrix}.$$

The goal is not only to find a solution to this system, but also to do so as efficiently as possible. It is impossible to reduce the size of these matrices without ignoring some of the sampled data or reducing the detail of the fitting spline. Reducing the complexity of C, thereby facilitating its factorization, is the only means available to lower the amount of work done.

Recall from Definition (2.12) that for any parameter point u_i at most r, the basis order, of the B-splines are non-zero. Thus a more concise definition for rows of C can be given based on l, the index of the interval $(\mu_l < u_i < \mu_{l+1}]$ containing the point u_i:

$$C_i = [\ \overbrace{0 \quad \cdots \quad 0}^{l} \quad \overbrace{B^r_{l-r+1}(u) \quad \cdots \quad B^r_l(u)}^{r} \quad \overbrace{0 \quad \cdots \quad 0}^{m-l-r}\ ]$$

Given that the sequence $\{u_i\}_{i=1}^s$ is non-decreasing, it follows that the sequence of interval indices $\{l\}_{i=1}^s$ is also non-decreasing. This strict ordering allows for the grouping of the evaluation points by the index of the interval in which they lie. This means that the corresponding basis evaluations will have the non-zero values in the same columns. The restriction that every basis function $B_i(u)$ is evaluated at least once forces each row $C_i, i > 1$, to have a least one non-zero column entry in common with its predecessor, C_{i-1}.

This allows the presentation of C as a block matrix with each block containing the evaluation of all the points in the same interval.

$$C = \begin{bmatrix}
\overbrace{}^{r} & & & & & & & & \\
x & x & x & x & & & & & \\
x & x & x & x & & & & & \\
x & x & x & x & & & & & \\
 & x & x & x & x & & & & \\
 & x & x & x & x & & & & \\
 & x & x & x & x & & & & \\
 & & x & x & x & x & & & \\
 & & & x & x & x & x & & \\
 & & & & x & x & x & x & \\
 & & & & x & x & x & x & \\
 & & & & & x & x & x & x \\
 & & & & & x & x & x & x \\
 & & & & & x & x & x & x \\
 & & & & & & x & x & x & x \\
 & & & & & & x & x & x & x \\
 & & & & & & x & x & x & x
\end{bmatrix} \begin{matrix} \\ \\ \\ u_i \in (\mu_0, \mu_1] \\ \\ \\ u_i \in (\mu_1, \mu_2] \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \tag{3.2}$$

The blocks will be denoted by superscripting C. Thus, the block containing entries for interval $(\mu_l, \mu_{l+1}]$ will be termed $C^{(l)}$. As an example, assume that there are j parameter values, ($u_i, \ldots u_{i+j-1}$), in the interval $(\mu_l, \mu_{l+1}]$. Evaluating the B-splines at each of the j parameter values yields the submatrix $C^{(i)} \in \mathbb{R}^{j \times r}$.

$$C^{(i)} = \begin{bmatrix}
B_{l-r+1}(u_i) & B_{l-r}(u_i) & \cdots & B_l(u_i) \\
B_{l-r+1}(u_{i+1}) & B_{l-r}(u_{i+1}) & \cdots & B_l(u_{i+1}) \\
\vdots & \vdots & & \vdots \\
B_{l-r+1}(u_{i+j-1}) & B_{l-r}(u_{i+j-1}) & \cdots & B_l(u_{i+j-1})
\end{bmatrix}$$

The objective is to produce an upper triangular matrix by eliminating, in a column-by-column fashion, the lower triangular elements. The total computation time is a function of the number of lower triangular elements and thus the goal is to minimize the number of such elements. This strict format of the matrix provides the opportunity to perform the factorization more efficiently.

3.5 Factorization

In producing an upper triangular matrix, the total computation time is primarily a function of the number of lower triangular elements. Therefore, the first goal is to minimize the number of such elements.

The fundamental methods used to solve a least squares system were given in Section 2.3. Now details will be given for the factorization method chosen, along with an outline of the basis matrix structure.

Various methods can be used to compute a QR-decomposition. Though each performs the job, there are particular classes of matrices to which each method is best suited. Givens Rotations are well suited to decomposing sparse matrices, as each rotation can eliminate a specific lower triangular element. The elements found in the B-spline fitting systems exhibit coherence across both rows and columns, requiring excessive numbers of rotation matrices. This large number of consecutive elements makes factorizations such as the Modified Gram-Schmidt method attractive. However, as demonstrated later, these systems require the mass introduction of zeros in well defined, consecutive locations. Householder transformations are well suited to this situation. Despite incurring more computational expense than the Modified Gram-Schmidt in general, Householder transformations will save time in this specific case of matrices with entries from B-spline evaluations. Numerical stability in the decomposition is also afforded by Householder transformations.

3.5.1 Methodology

Definition 3.5 *A* **Householder transformation**, $H \in \mathbb{R}^{s \times s}$ *is an orthogonal,*

symmetric matrix defined by a vector $\vartheta \in \mathbb{R}^s$ *and the formula*

$$H = I - 2\vartheta\vartheta^T/\vartheta^T\vartheta$$

.

A complete description of how to find the Householder transformations, together with proofs that they represent reflections and that they are both orthogonal and symmetric, can be found in [18].

The most relevant attribute of these transformations is that for a given column vector $C^i \in \mathbb{R}^s$ the application, via matrix pre-multiplication, of a Householder transformation is simply the reflection of C^i within the hyper-plane $span\{\vartheta\}^\perp$. Thus, the 2-norm of C^i is unchanged by the transformation, preserving the residual as required.

An appropriately constructed vector ϑ (see [18]) ensures that the resulting Householder matrix will reflect C^i in such a way that all its elements, except the first, will become zero:

$$H_i C^i = H_i[C_{i,1}, \cdots, C_{i,s}]^T = [C'_{i,1}, 0, \cdots, 0]^T$$

The target vectors, $\{C_j\}_{j=1}^m$, are chosen to contain all lower triangular elements of successive columns from the basis value matrix, $C^j = \{C_j\}_{j=i}^s$. The application of a Householder transformation, derived individually for each of the target vectors, will introduce zeros in all positions that lie below the diagonal. Once a target vector from each column has been extracted and transformed, the basis value matrix, C, will be in upper-triangular, or simply *triangular* form.

Throughout this process, preservation of the Euclidean norm means that

$$C_{i,i}^2 + \cdots + C_{i,s}^2 = (C'_{i,i})^2$$

so

$$C'_{i,i} = \pm\sqrt{C_{i,i}^2 + \cdots + C_{i,s}^2}.$$

3.5.2 Solving systems with Householder Transformations

Beginning with a simple overdetermined system

$$CV = D, /'C \in I\!R^{s\times m}, /'s \geq m.$$

The factorization process involves finding a sequence of matrices, $H_1, \ldots, H_m$, which premultiply C, transforming it to an upper triangular matrix. Understanding the factorization process is simplified by the following example, extended from [18].

Example 3.2 *Suppose there are $s = 6$ data points and the chosen fitting curve has order $r = 4$ and dimension $m = 5$, implying the curve has two intervals. This gives a basis matrix $C \in I\!R^{6\times 5}$.*

Assume that the Householder matrices H_1 and H_2 have been computed so that

$$H_2H_1C = \begin{bmatrix} x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & \mathbf{x} & x & x \\ 0 & 0 & \mathbf{x} & x & x \\ 0 & 0 & \mathbf{x} & x & x \\ 0 & 0 & \mathbf{x} & x & x \end{bmatrix}.$$

The third column, C^3, is the current target vector. It is to be reflected in a manner that zeros all elements below the diagonal. Focusing on the boldface entries, the

matrix $\tilde{H}_3 \in \mathbb{R}^{4\times 4}$ *must be found to produce*

$$\tilde{H}_3 \begin{bmatrix} \mathbf{x} \\ \mathbf{x} \\ \mathbf{x} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} x \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

If $H_3 = diag(I_2, \tilde{H}_3)$, *that is to say*

$$\left[\begin{array}{c|c} I_2 & 0 \\ \hline 0 & \tilde{H}_3 \end{array}\right], \tag{3.3}$$

then

$$H_3H_2H_1C = \begin{bmatrix} x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & x & x \end{bmatrix}.$$

After two more steps, an upper triangular matrix is obtained and can be used to solve the system.

3.5.3 Optimizing the Process

Typically a trade-off must be made between the saving of time and the saving of space. The special format of C and the decomposition method combine to allow savings in both areas and in two different ways.

Reducing Computational Complexity

The decomposition process, as detailed so far, does not take into account the special format of C given in equation (3.2). The potential optimization to the fitting process should be obvious with the following extension to the last example.

Example 3.3 *Suppose that the evaluation points* $\{u_i\}_{i=1}^{s}$ *for this problem have two points in the first knot interval and the remaining four in the second.*

The basis matrix is now:

$$C = \begin{bmatrix} B_1(u_1) & B_2(u_1) & B_3(u_1) & B_2(u_1) & 0 \\ B_1(u_2) & B_2(u_2) & B_3(u_2) & B_4(u_2) & 0 \\ 0 & B_2(u_3) & B_3(u_3) & B_4(u_3) & B_5(u_3) \\ 0 & B_2(u_4) & B_3(u_4) & B_4(u_4) & B_5(u_4) \\ 0 & B_2(u_5) & B_3(u_5) & B_4(u_5) & B_5(u_5) \\ 0 & B_2(u_6) & B_3(u_6) & B_4(u_6) & B_5(u_6) \end{bmatrix}. \tag{3.4}$$

When determining the first Householder transformation to zero out all entries below the diagonal, a reflection satisfying

$$H_1 \begin{bmatrix} B_1(u_1) \\ B_1(u_2) \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} B_1(u_1) \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

is needed. This problem can be reduced to finding

$$\tilde{H}_1 \begin{bmatrix} B_1(u_1) \\ B_1(u_2) \end{bmatrix} = \begin{bmatrix} B_1(u_1) \\ 0 \end{bmatrix}.$$

A new attribute of the B-spline matrix is the *column extent.* The column extent indicates the last non-zero basis value and is based on both the data points, $\{\alpha_i\}_{i=1}^{s}$, and the evaluation points, $\{u_i\}_{i=1}^{s}$.

Definition 3.6 *The* **column extents**, $\{e\}_{i=1}^{m-r+1}$ *are values such that* e_i*, which is based on the* i*-th column of a B-spline matrix, is the largest value* j *such that* $B_i(u_j)$ *is non-zero.*

For example, using the matrix in (3.4) e_1 is 2 since $B_1(u_2)$ is the last non-zero entry in the column. Similarly, the column extents of the remaining columns is 6.

The existence of column extents allows us to make the following observation; while the Householder transformation used to introduce zeros to i-th column is $\tilde{H}_i \in I\!R^{s-i\times m-i}$ in general, the special format of C reduces the extent of non-zero entries of the column vector C^i to $s-e_i$. This reduces the size of the transformation matrix for the i-th, so $\tilde{H}_i \in I\!R^{s-e_i\times m-i}$.

Formally, a refinement to equation (3.3) yields:

$$H = diag(I_i, \tilde{H}_i, I_{s-e_i}).$$

This reduction in size means that not only is $\tilde{H}_i$ less costly to compute but, as will be show later, the number of *shadow columns* that need to be computed is also reduced.

Another valuable set of information is the set of *column tops.* These indices indicate the first non-zero entries in each column vector, C^i. They are used further on in the discussion of optimizations and are defined procedurally.

Definition 3.7 *Initialize the set of* **column tops**, $\{\tau_i\}_{i=1}^{m}$*, to their maximum possible value,* s*. For each of the data points* $\{u_i\}_{i=1}^{s}$*, and its corresponding interval*

index, l_i raise the column tops for each of the columns with non-zero basis value to be at least the current row index, i. So $\{\tau_j\}_{j=l_i}^{l_i+r} = min(i, \tau_j)$,

Reducing Storage Requirements

There are two potential areas in which storage requirements may be reduced. The first is based on the column extents, $\{e_i\}_{i=1}^{m}$, described in the previous section. This provides an immediate savings of

$$\sum_{i=1}^{n}(s - e_i).$$

These savings are based exclusively on elements below the diagonal. A second avenue towards saving space is provided through the use of *shadow columns.* To illustrate this concept, start with the basis value matrix

$$C = \begin{bmatrix} b & \mathbf{b} & \mathbf{b} & 0 & 0 \\ b & \mathbf{b} & \mathbf{b} & 0 & 0 \\ 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} & 0 \\ 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} & 0 \\ 0 & 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} \\ 0 & 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} \end{bmatrix}. \tag{3.5}$$

To indicate the various states of the matrix entries during the transformations distinct typefaces are used. The original B-spline values are indicated by $\mathbf{b}$, b denotes lower triangular entries of the column being processed and $\mathbf{b}$ marks elements that will be modified as the Householder transformation is applied to that column. Elements originally known to be zero and previously modified will be indicated by β. This helps demonstrate the changes incurred from the transformation.

Having found a Householder transformation that reduces the italic elements to zero, all columns that have non-zero entries above the lowest italic entry must be premultiplied. Again, the affected elements are denoted by the sans-serif **b**.

$$C = \begin{bmatrix} \mathbf{b} & \mathbf{b} & \mathbf{b} & 0 & 0 \\ 0 & b & \mathbf{b} & 0 & 0 \\ 0 & b & \mathbf{b} & \mathbf{b} & 0 \\ 0 & b & \mathbf{b} & \mathbf{b} & 0 \\ 0 & 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} \\ 0 & 0 & \mathbf{b} & \mathbf{b} & \mathbf{b} \end{bmatrix} \tag{3.6}$$

The first transformation failed to alter the known zero elements. Those locations changed as a result of the second Householder transformation, again shown as the sans-serif entries, will include elements previously unused. Continuing with the third transformation, there are again elements that are changed to become non-zero.

$$C = \begin{bmatrix} \mathbf{b} & \mathbf{b} & \mathbf{b} & 0 & 0 \\ 0 & \mathbf{b} & \mathbf{b} & \beta & 0 \\ 0 & 0 & b & \mathbf{b} & \mathbf{0} \\ 0 & 0 & b & \mathbf{b} & \mathbf{0} \\ 0 & 0 & b & \mathbf{b} & \mathbf{b} \\ 0 & 0 & b & \mathbf{b} & \mathbf{b} \end{bmatrix} \tag{3.7}$$

Once the transformations reach the leftmost column, the *shadow columns* will not modify any additional zero entries. Householder transformations continue to be applied to each column until only upper triangular elements remain. Thus, the

example produces the final matrix:

$$C = \begin{bmatrix} \mathbf{b} & \mathbf{b} & \mathbf{b} & 0 & 0 \\ 0 & \mathbf{b} & \mathbf{b} & \beta & 0 \\ 0 & 0 & \mathbf{b} & \mathbf{b} & \beta \\ 0 & 0 & 0 & \mathbf{b} & \beta \\ 0 & 0 & 0 & 0 & \mathbf{b} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.8}$$

With the example complete, the notion of *shadow columns* can be formally defined, making the column top values, $\{\tau_i\}_{i=1}^{m}$, essential.

Definition 3.8 *When applying the i-th Householder transformation to the matrix C the* **shadow columns** *are the columns $C^{i+1}, \ldots, C^{j}$, where j is the minimum value for which the column extent $\tau_i < e_j$.*

Thus, shadow columns are computed for each column that has a non-zero entry with an index higher than the lowest non-zero entry of the current target vector.

Example 3.4 *Returning to the previous example, the matrix in (3.5) has a target vector of C^1. It has as shadow columns C^2, C^3, since $e_1 < \tau_4$. Similarly, the matrix in (3.6) has C^2 as its target vector, with C^3, C^4 as its shadow columns since $e_2 < \tau_5$. The last matrix (3.7) has shadow columns, C^4, C^5.*

Once the extents of all the shadow columns are computed, those matrix elements that are unaffected by the transformations are known. These are the upper triangular entries which remained zero from (3.5) to (3.8). As these entries do not contribute to the factorization, there is no need to store them.

Depending on the savings provided by both the column extents and shadow column methods, it may be worthwhile to implement some form of sparse matrix (see Chapter 7). The space savings must be weighted against the increased cost of indexing such a structure.

3.6 Fitting to Multidimensional Data

The fitting procedure from the previous section provided a method for determining a control mesh, $V = \{\nu_i\}_{i=1}^{m}$, for an underlying set of B-splines, $B_i^r(u)$, from a set of data points, $D = \{d_i\}_{i=1}^{s}$. This relationship can be expressed as

$$D \xrightarrow{B_i^r} V. \tag{3.9}$$

We can use the same set of B-splines to find fits between multiple data sets $\{D_i\}_{i=1}^{n}$ and their computed control vectors $\{\nu_i\}_{i=1}^{n}$, yielding

$$D_i \xrightarrow{B_i^r} V_i.$$

Having a one multidimensional data set D^n accommodates the usage of the curve fitting process on each of the elements of the slice set Θ in a particular dimension.

Example 3.5 *Taking the data set $D \in \mathbb{L}^{5\times4\times6}$ in Example 2.2 and generating the slice set in the second dimension would produce the slice set $\Theta = \{\theta_i\}_{i=1}^{4}$, where $\theta_i \in \mathbb{L}^{5\times4}$. Using the curve fitting process the sets of control vertices $\{\nu_i\}_{i=1}^{4}$ could be produced.*

In general, when given the data points $D \in \mathbb{L}^{i_1,\cdots,i_n}$ and a dimension j, $1 < j < n$, in which to perform the fits, there are

$$\prod_{l=1}^{n} i_l/i_j$$

elements in the slice set

$$\Theta = \{\{d_{i_1,\cdots,i_n}\}_{i_j=1}^{s_j}\}_{i_k}^{s_k}\}_{k=1,\cdots,j-1,j+1,\cdots,n}. \tag{3.10}$$

Now a fit can be performed on each of the elements of the constructed slice set

$$D_\theta \xrightarrow{B_i^r} V_\theta, \theta \in \Theta.$$

Each of the slices in equation (3.10) has s_j elements and each slice produces m_j control points. This means that the original data $D \in \mathbb{L}^{i_1 \cdots i_n}$ is approximated by the B-splines B_i^r with the control points $V \in \mathbb{L}^{i_1 \times \cdots \times i_{j-1} \times m \times i_{j+1} \times \cdots \times i_n}$.

Chapter 4

Fitting in Arbitrary Dimensions

Until now, discussion has been restricted to problems that can be solved with either a one-variable B-spline i.e. a curve, or a set thereof, sharing a common set of basis functions. In this chapter, the need for extending the fitting scheme to include multi-variable splines, or *hyper-splines*, is presented. Increasing the number of variables in the fitting splines allows the fitting of entire multidimensional data sets with a single B-spline formulation. This leads to an algorithm that produces approximating B-spline hyper-surfaces as a generalization of the existing curve fitting process.

4.1 Motivation

While the example of fitting colour data in Chapter 3 was instructive, its simplicity reflects the fact that it is unrealistic in several aspects. The most essential shortcoming is that there is little interest in increasing the accuracy of single colour printing. Existing calibration methods for monochrome printers are sufficient [1];

demands for fidelity seldom exceed capabilities. A second reason why such simple fitting techniques are inadequate is that no one-dimensional colour space exists. The common models, as outlined in Section 2.2, all have at least three components. The only single component model that can be considered is one based upon the wavelength of light. This model is also inadequate, as no existing physical device is capable of producing the entire spectrum of colours based on a single input. Instead, a desired colour is separated into components such as red, green and blue, which can be output by electron guns.

The last motivation to move beyond splines of one variable comes from the intermixing of colours. For example, magenta inks may have traces of yellow in them or combinations of magenta and cyan may reflect more blue than is contained in each of the individual inks. It is this final point that is of primary concern. Ideally, neither of the proposed infidelities exists in the printer inks. Realistically, only a spectroradiometer can determine the resultant output colour to any given input; therefore, the inputs cannot be examined in isolation. Using simple, one variable, fitting splines denies the opportunity to reflect this intermixing of colours. Evaluation can only be performed along the data dimension in which the approximation was computed. A means of extracting information about the printer's results in each of the colour dimensions is required. Providing a multi-variable spline permits the necessary interpolation for intermediate data points. Thus, points not on any particular axis can be evaluated.

Data sets for colour consist of elements that have both the desired colour and the actual output colour, as measured by the spectroradiometer. This leads to the expected conclusion that the colour data sets are not simple vectors, nor are they matrices, which could be addressed satisfactorily by the fitting techniques in [32, 16]. Instead, the data comes as rectangular lattice with data dimension three, four, or

higher. It would have been a simple matter to extend Srećković's fitting schemes [32] to accommodate data sets whose dimension is higher than two. However simplicity is outweighed by the work involved in customizing the application for each separate data dimension. A fitter was needed for the quick handling of colour spaces with unforeseen data dimensions. Additionally, the increased precision of the resulting fits of Forsey and Bartels [16] was not required. As will be shown later, the data is well approximated by basic tensor product B-splines.

4.2 Notation

With the objective of building on the existing B-spline fitter, the single-variable B-spline curve from Chapter 3 is extended to produce a *B-Spline Tensor Product Hyper-Surface.*

Definition 4.1 *A* **B-Spline Tensor Product Hyper-Surface**, $C(u^1, u^2, \ldots, u^n)$, *is an n-variable spline defined by*

$$C(u^1, u^2, \ldots, u^n) = \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_n=1}^{m_n} \nu_{i_1, i_2 \cdots i_n} B_{i_1}^{r_1}(u^1; \mu_1) B_{i_2}^{r_2}(u^2; \mu_2) \cdots B_{i_n}^{r_n}(u^n; \mu_n) \tag{4.1}$$

where $B_{i_j}^{r_j}(u^j; \mu_j)$ is the j-th B-spline basis, with order r_j, dimension m_j, and is defined over the knot sequence $\mu_j = \{\mu_{j,i}\}_{i=0}^{m_j+r_j+1}$. The control points $V = \nu_{i_1, i_2, \cdots, i_n} \in \mathbb{E}_t^{m_1 \times m_2 \times \cdots \times m_n}$, form a rectangular lattice.

The reader may recall from Example 2.2 that the j-th basis functions in the i-th variable were denoted as $B_j^{(i)}(\delta^i)$. At that point the extra detail provided by the knot sequences was unnecessary. Now, the superscript (i) was replaced by the

argument μ_i to reinforce the notion that the B-splines in each variable are independent of those in the remaining variables. That stated, from this point onward the short form $B_{i_j}^{r_j}(u^j)$ will be used in place of $B_{i_j}^{r_j}(u^j;\mu_j)$ with the understanding that the B-splines in each of the variables can have a unique knot sequence.

The multidimensional data set describing the n-variable mapping problem consists of a set of inputs $\delta^1,\ldots,\delta^n$ and the result components $\alpha^1,\ldots,\alpha^t$. Each input is sampled at various points along a range, giving $low^i \leq \delta^i$ in the i-th input. Additionally, each input has sampling point at each extent, giving:

$$\delta^i = low^i \Rightarrow \delta_1^i$$

$$\delta^i = high^i \Rightarrow \delta_{n_i}^i.$$

A single sampling set produces a data set with a data dimension of one. When observations are based on multiple inputs the set will contain one entry from each of the possible combinations of inputs. The entire set of combinations can be expressed through the Cartesian product of the individual sampling sets. This provides a data set with rectangular lattice structure whose input components are the n-tuples $(\delta_{i_1}^1,\ldots,\delta_{i_n}^n)$, subject to the ranges detailed previously. The corresponding output component is

$$\alpha_{i_1,\ldots,i_n} = (\alpha_{i_1,\ldots,i_n}^1,\ldots,\alpha_{i_1,\ldots,i_n}^t).$$

So the fitting problem is one of finding a spline of the form

$$C(\delta) = C(\delta^1,\ldots,\delta^n) = \sum_{i_1}^{m_n}\cdots\sum_{i_1}^{m_1}\nu_{i_1,\ldots,i_n}B_{i_1}^{r_1}(\delta^i)\cdots B_{i_1}^{r_1}(\delta^i),$$

where $\nu_{i_1,\ldots,i_n} = (\nu_{i_1,\ldots,i_n}^1\nu_{i_1,\ldots,i_n}^t)$.

In the multidimensional generalization the evaluation points continue to be identical to the input points. So in the i-th spline variable the evaluation points are $\delta^i = \{\delta_j^i\}_{j=1}^{s_i}$, meaning the fitting spline should satisfy $C(\delta_{i_1}^1,\ldots,\delta_{i_n}^n) \approx \alpha_{i_1,\ldots,i_n}$.

Extending the residual function from Equation (3.2) is now straight-forward.

Definition 4.2 *The* **Least Squares Residual Norm** *of a data set* $D \in I\!L_{n+t}^{s_1 \times s_2 \times \cdots \times s_n}$ *and a fitting hyper-spline* $C(u^1, u^2, \cdots, u^n)$ *that approximates it is*

$$res(C, D) = \sum_{i_1=1}^{s_1} \sum_{i_2=1}^{s_2} \cdots \sum_{i_n=1}^{s_n} \| C(\delta_{i_1}^1, \delta_{i_2}^2, \ldots, \delta_{i_n}^n) - \alpha_{i_1, i_2, \ldots, i_n} \| \tag{4.2}$$

where $\alpha_{i_1, i_2, \ldots, i_n} = (\alpha_{i_1, i_2, \ldots, i_n}^1, \ldots, \alpha_{i_1, i_2, \ldots, i_n}^t)$.

The definition of a curve fitter must also be extended to arbitrary dimensions.

Definition 4.3 *A* **hyper-surface fit** *provides the solution to the minimization of equation (4.2).*

With these definitions in place, the process of fitting data in arbitrary dimensions can be detailed.

4.3 Fitting Algorithm

In Chapter 3 the control net, $V \in \mathbb{L}_t^m$, was found for a curve that fit a vector of results points, $A \in \mathbb{L}_t^s$. Such problems had a data dimension of one, a spline domain dimension of one, and t result components. The equivalence between the data and spline domain dimensions is fundamental and always exists. Problems of increased complexity, such as colour data fitting, involve finding the control net $V \in \mathbb{L}_t^{m_1 \times \ldots \times m_n}$ for a hyper-surface that approximates the lattice of result components $A \in \mathbb{L}_t^{s_1 \times \ldots \times s_n}$. Here, both the spline domain dimension and data dimension are n.

When dealing with multidimensional data sets the fitting of each element of the slice set along the j-th data dimension yields a control net with m_j elements. With this reduction, focus moves from the result components, $A \in \mathbb{L}^{s_1 \times \cdots \times s_n}$, to a data set

$$\omega_{i_1,i_2,\ldots,i_n} \in \mathbb{L}^{s_1 \times \cdots \times s_{j-1} \times m_j \times s_{j+1} \times \cdots \times s_n}. \tag{4.3}$$

Entries in the reduced data set, $\omega_{i_1,i_2,\ldots,i_n} = (\omega^1_{i_1,i_2,\ldots,i_n}, \ldots, \omega^t_{i_1,i_2,\ldots,i_n})$ retain the same number of result components as the original.

In general there is no ordering of data dimensions that, when slicing is performed, provide results that are superior to any other order. For convenience the convention of taking slices from the highest data dimension, n, through to the lowest, 1, is used. This can also be described as working from the n-th spline variable, u_n, through to the first spline variable, u_1. If, as suggested, only one slice is taken, the convention implies that it would be in the n-th data dimension. Thus, $j = n$, simplifying equation (4.3) to:

$$\omega_{i_1,i_2,\ldots,i_n} \in \mathbb{L}^{s_1 \times \cdots \times s_{n-1} \times m_n}. \tag{4.4}$$

With the ordering of an approach in place, the fitting process is performed in

a stepwise fashion over each of the n variables in the data dimension. The last preparatory item is the set of substitution variables $\{\omega^{(k)}_{i_{n-k+1},\ldots,i_n}\}_{k=1}^{n}$, referred to as the *intermediate control nets.*

The process begins when the first intermediate control net is extracted from the original hyper-spline formula

$$C(u_1,\cdots,u_n) = \sum_{i_n=1}^{m_n} \underbrace{\sum_{i_{n-1}=1}^{m_{n-1}} \cdots \sum_{i_1=1}^{m_1} \nu_{i_1,\cdots,i_{n-1},i_n} B^{r_1}_{i_1}(u_1)\cdots B^{r_{n-1}}_{i_{n-1}}(u_{n-1})}_{\omega^{(n)}_{i_n}} B^{r_n}_{i_n}(u_n), \tag{4.5}$$

where $\omega^{(n)}_{i_n} = (\omega^{1^{(n)}}_{i_n},\ldots,\omega^{t^{(n)}}_{i_n})$. By rewriting equation (4.5) in terms of $\omega^{(n)}_{i_n}$, the fitting spline is simplified to:

$$C(u_1,\cdots,u_n) = \sum_{i_n=1}^{m_n} \omega^{(n)}_{i_n} B^{r_n}_{i_n}(u_n)$$

In turn, the fitting problem reduces to solving the system:

$$\sum_{i_n=1}^{m_n} \omega^{(n)}_{i_n} B^{r_n}_{i_n}(u_n) = A. \tag{4.6}$$

Since A is extracted directly from D, its entries are $\alpha_{i_1,\ldots,i_n}$, with components $(\alpha^1_{i_1,\ldots,i_n}, \ldots, \alpha^t_{i_1,\ldots,i_n})$, where $1 \le i_j \le s_j$; $j = 1,\ldots,n$.

The hyper-spline fitting problem is now expressed in the same format as Equation (3.9). The solution to such systems can be found by taking a slice set of the data and performing a series of curve fits, as described in Section 3.6. With $A \in \mathbb{R}_t^{s_1\times\ldots\times s_n}$ and the reduction from equation (4.4) the structure of the first intermediate control net is known to be $\omega^{(n)}_{i_n} \in \mathbb{R}_t^{s_1\times\ldots\times s_{n-1}\times m_n}$.

The original problem of finding $V \in \mathbb{R}^{m_1\times m_2\times\cdots\times m_n}$ has been reduced to the simpler task of finding a hyper-spline in $n-1$ variables. The next step in the

process is to solve for $\omega_{i_n}^{(n)}$. This leads to another approximation:

$$\sum_{i_{n-1}=1}^{m_{n-1}} \underbrace{\sum_{i_{n-2}=1}^{m_{n-2}} \cdots \sum_{i_1=1}^{m_1} C_{i_1,i_2\cdots i_n} B_{i_1}^{r_1}(u_1) \cdots B_{i_{n-2}}^{r_{n-2}}(u_{n-2})}_{\omega_{i_{n-1},i_n}^{(n-1)}} B_{i_{n-1}}^{r_{n-1}}(u_{n-1}) = \omega_{i_n}^{(n)},$$

Now the substitution is performed, as in Equation (4.6), to get

$$\sum_{i_{n-1}=1}^{m_{n-1}} \omega_{i_{n-1},i_n}^{(n-1)} B_{i_{n-1}}^{r_{n-1}}(u_{n-1}) = \omega_{i_n}^{(n)}. \tag{4.7}$$

With the intermediate control points $\omega_{i_n}^{(n)}$ available from equation (4.6), the solution to (4.7) is also given by equation (3.9).

The substitution process continues by solving the general equation

$$\sum_{i_{k-1}=1}^{m_{k-1}} \underbrace{\sum_{i_{k-2}=1}^{m_{k-2}} \cdots \sum_{i_1=1}^{m_1} \nu_{i_1,i_2\cdots i_n} B_{i_1}^{r_1}(u_1) \cdots B_{i_{k-2}}^{r_{k-2}}(u_{n-1})}_{\omega_{i_{k-1},\ldots,i_n}^{(k-1)}} B_{i_{k-1}}^{r_{k-1}}(u_{k-1}) = \omega_{i_k,\ldots,i_n}^{(k)}$$

for $k = n-1, \ldots, 1$. When the step n=1 is solved the fitting is complete. The unknown control net from equation (4.5) is found in the identity

$$\nu_{i_1,\ldots,i_n} \equiv \omega_{i_1,\ldots,i_n}^{(1)}. \tag{4.8}$$

Example 4.1 *The data set $D \in \mathbb{L}^{5\times4\times6}$, from Example 2.2, is to be fit by a 3-variable spline, $C(u_1, u_2, u_3)$, with basis dimensions $m_1 = m_2 = m_3 = 3$. Following the stated convention, the slice set from the third dimension is taken first. The 5*4=20 6-element vectors are passed through the curve fitter to produce twenty 3-element vectors of intermediate control points. After this step the result components of the data set $D \in \mathbb{L}^{5\times4\times6}$ are replaced by $\omega_{i_3}^{(3)} \in \mathbb{L}^{5\times4\times3}$. Continuing, by slicing along the second data dimension, yields 5*3=15 4-element vectors. These are given to the curve fitter, which computes fifteen 3-element vectors producing*

$\omega^{(2)}_{i_2,i_3} \in \mathbb{L}^{5\times 3\times 3}$. *Finally, the first data dimension is sliced, providing 3*3=9 5-element vectors for the curve fitter. The nine resulting 3-element vectors define* $\omega^{(1)}_{i_1,i_2,i_3} \in \mathbb{L}^{3\times 3\times 3}$. *From the identity in (4.8) the control points* $\nu_{i_1,i_2,i_3} \equiv \omega^{(1)}_{i_1,i_2,i_3}$ *produce* $V \in \mathbb{L}^{3\times 3\times 3}$, *as desired.*

4.4 Multidimensional data

To this point no justification of the suitability of this fitting process has been offered. There are several assumptions that have been made which will now be explained.

The first assumption is that the control net provided by this stepwise process is the best one. There are two underlying questions that ground this assumption. First, does a direct, non-stepwise method for such fitting exist; and if so, do the results provided by this method match those obtained from the stepwise method?

Indeed a direct method does exist, but as the following discussion demonstrates its computational complexity makes its use prohibitively expensive. To demonstrate this expense, two notational conveniences are used; the first is the function $\Psi(C)$, which yields the column vector obtained by placing the columns of C, from right to left, atop one another. The second is the *Kronecker product* of two matrices $C_{(1)}$ and $C_{(2)}$, denoted by $(C_{(1)} \otimes C_{(2)})$.

Definition 4.4 *The* **Kronecker product** *of two matrices* $C_{(1)} \in \mathbb{R}^{s_1 \times m_1}$ *and* $C_{(2)} \in \mathbb{R}^{s_2 \times m_2}$ *is the* $s_1 s_2 \times m_1 m_2$ *matrix given by*

$$(C_{(1)} \otimes C_{(2)}) = \begin{bmatrix} C_{(1)1,1}C_{(2)} & C_{(1)1,2}C_{(2)} & \dots & C_{(1)1,m_1}C_{(2)} \\ C_{(1)2,1}C_{(2)} & C_{(1)2,2}C_{(2)} & \dots & C_{(1)2,m_1}C_{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ C_{(1)s_1,1}C_{(2)} & C_{(1)1,2}C_{(2)} & \dots & C_{(1)s_1,m_1}C_{(2)} \end{bmatrix}.$$

Finding the least squares solution to the system

$$\sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \nu_{i_1,i_2} B^{r_1}_{(1)i_1} B^{r_2}_{(2)i_2} = D_{s_1,s_2}$$

can be reduced to

$$(C_1 \otimes C_2)\Psi(V) = \Psi(D). \tag{4.9}$$

This is presented without proof, with the reader referred to the literature [11, 12] for details. Equation (4.9) demonstrates that a multi-variable fitting spline can be computed directly. The cost of such a direct approach is dominated by the expense of the factorization of $(C_1 \otimes C_2)$. Since factorization is an $O(n^3)$ process, the cost of solving the minimization directly would be

$$(s_1 s_2 m_1 m_2)^3$$

compared with a cost of

$$(s_1 m_1)^3 + (s_2 m_2)^3$$

for the stepwise method. The potential savings offered by the iterative method means that it is almost always chosen in the literature, as was done here.

The second assumption is that the stepwise process can be applied to the dimensions of the data and intermediate control nets in any order; the convention used takes slices starting in the highest data dimension and progressing towards the first data dimension. This assumption is valid because all work is being performed in Cartesian coordinates; meaning that the coordinate vectors are mutually orthogonal. Thus, minimizing the sum of squares of all the components of a vector can be achieved by minimizing each of the coordinates individually, without adjusting any of the others. The mathematical equivalence of the iterative and direct methods, also detailed in [11, 12], means that no ordering of the iterations will provide a better solution; so any order will suffice.

The final assumption was that the control net could be found independently for each dimension of the space in which the fitting spline exists. The α-bases are found by a SVD, as outlined in Section 2.2. This produces a basis that is not only linearly independent, but also orthogonal. Least squares fitting and orthogonality

are mutually supportable; implying that the least squares minimization, with respect to an orthogonal basis, can be carried out independently with respect to each basis element. Such characteristics are not demonstrated by non-orthogonal bases.

Chapter 5

Visualization

In Chapter 3 the approximation criterion was presented as a function. This provides a simple analytic tool to measure the success with which a data set is fit by a curve.

When dealing with colour data there are other, less objective measures of success. These are based on known attributes of the data set. For example, with CMYK inputs, it is known that the first alpha basis will decrease as the percentage concentration of any of the input colours decreases. Such characteristics would help indicate problems not hinted at by the residual value. An example of this would be when the residual was not undesirably large while the hyper-surface approximation of the first alpha basis was constant, instead of decreasing. Another example comes from the knowledge that observations of inputs that are to produce dim colours, such as browns and dark blues, cannot be measured with the same accuracy as brighter colours. Thus, a high residual may be acceptable, provided that the majority of its size comes from poor approximation of such murky colours.

With these additional measures of *fit quality* established a means must be provided which allows the researcher to gauge these subjective criteria. The goal of this

work is to provide an easy-to-use and extensible set of tools that allow researchers to quickly develop applications. The resulting applications can be used not only to produce fits but also to help develop a satisfactory definition of quality through visualization of those fits.

The two constituents of the data set each present visualization problems. The inputs, or independent variables, are easily handled when they are not too numerous. In fitting with a planar curve, the results can be plotted as heights along a single axis. When dealing with two inputs, the results, plotted as heights above their arguments, require a projection for display by existing printers or monitors. Displaying the entire set of results as heights cannot be extended beyond this point. With three inputs, a domain volume may be drawn and the heights displayed for any slice that holds one of the inputs constant. The observer may be aware of the existence of a third input but is restricted to viewing a subset at any given moment. When faced with data sets containing more than three inputs only a subset of the domain can be displayed by the volume. Further, as in the three input case, only results from two of the chosen three inputs can be displayed.

The result components also introduce difficulties. Take for example where the number of output components was constant, $t = 8$, for data sets with varying numbers of inputs. Examining the result components for a one variable data set can be accomplished by plotting eight curves, one for each result component. It has been established that two independent variables is the limit to the height field method, regardless of whether they come from two inputs or from more inputs, with some being held constant. Thus, a series of eight surfaces can be used for data sets with multiple inputs. With a large number of result components, such as the eight alpha basis coefficients, the surfaces can become intertwined, leaving the option of displaying them as planes with contour lines, or simply viewing one

result component at a time.

Visualizing multidimensional data sets is an area of active research. Experimental methods for the display of results using motion, colour, or specialized data entry markers do permit the interpretation of entire data sets. Given the infancy of such work and the lack of applications that adequately address multidimensional data, the results from this work were analyzed by viewing subsets thereof with a commercial product: Wavefront's Data Visualizer.

5.1 Data Views

The methods used to present the fitting results are now introduced with the data set itself, providing the opportunity to demonstrate the subset techniques described earlier.

The visualization application restricts the number of input dimensions to three. With data sets having four sampling dimensions a tradeoff must be made to display static images. In this and subsequent visualization sections, the input for black, K, is considered to always be zero. This value was chosen for simplicity and does not reflect any particular bias in the result components. Outlining the domain serves as an indicator to the viewer that additional, unseen, data exists.

Using a mesh to display the data provides a reminder that it contains discrete points. This convention will allow the reader to disambiguate immediately between the data and the fitting surface, which is displayed through continuous shading.

Each figure contains three *cut planes* [36] through the CMY0 input volume. Interactively, only one plane is used and it is easily moved along the yellow axis. With each input having the range (0,255), the results at Y=0, Y=127.5 and Y=255

have been chosen to give the reader an impression of the data's characteristics.

Little is gained by providing displays for all eight of the result components. Rather, a select few components are used to demonstrate high level characteristics, such as constant value areas, or strictly decreasing regions, which should be mimicked by the fitting spline.

To accentuate the subtleties in the data, the range for each alpha component is different. The ranges will remain fixed across subsequent figures that display the same component.

Figure 5.1 shows the first result component, α^1. These coefficients exhibit the characteristic of decreasing value as concentrations of magenta and cyan increase. Again, it is essential that the fitting curve approximate this downward slope.

A display of the result component α^7 (Figure 5.2) provides a demonstration of aberrations in the data, seen as peaks in the topmost slice. These are typically due to experimental error, but they will still contain some information about the colour produced by the inputs at the given concentrations. It is the fitter's task to ignore the error and the investigator's job to provide a spline with enough precision to interpolate the correct value.

The final data figure (Figure 5.3) shows α^3, which is unusual in that the coefficients tend to form a valley in the mid-range of cyan. This last figure emphasizes the independence of both the shape and range of the alpha basis coefficients.

5.2 Fitting Surfaces

The fitting spline exists, necessarily, over the same domain as the data set. Given the continuity of the specification, it may be sampled as frequently as necessary,

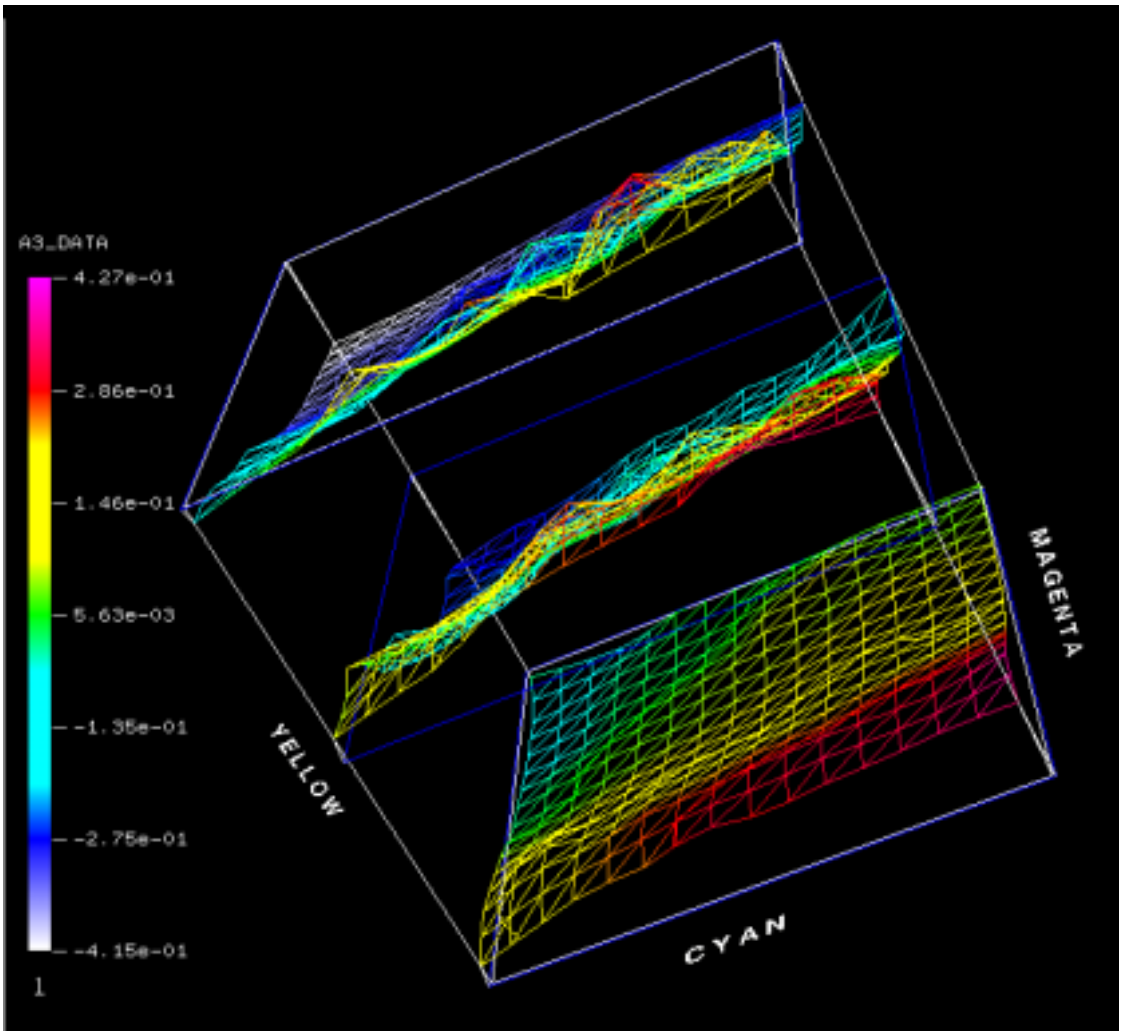

Figure 5.1: Cut planes for the α^3 coefficient.

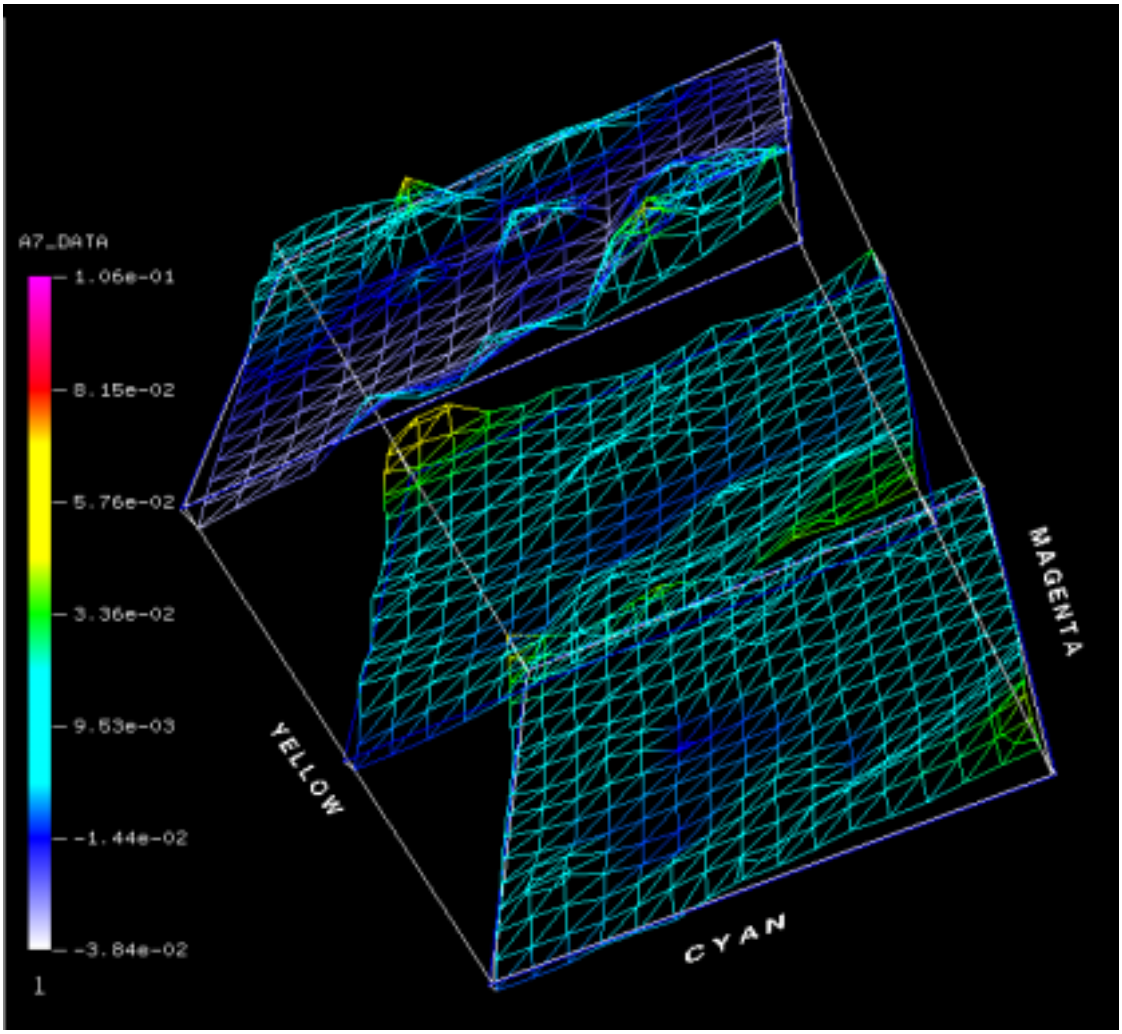

Figure 5.2: Cut planes for the α^7 coefficient.

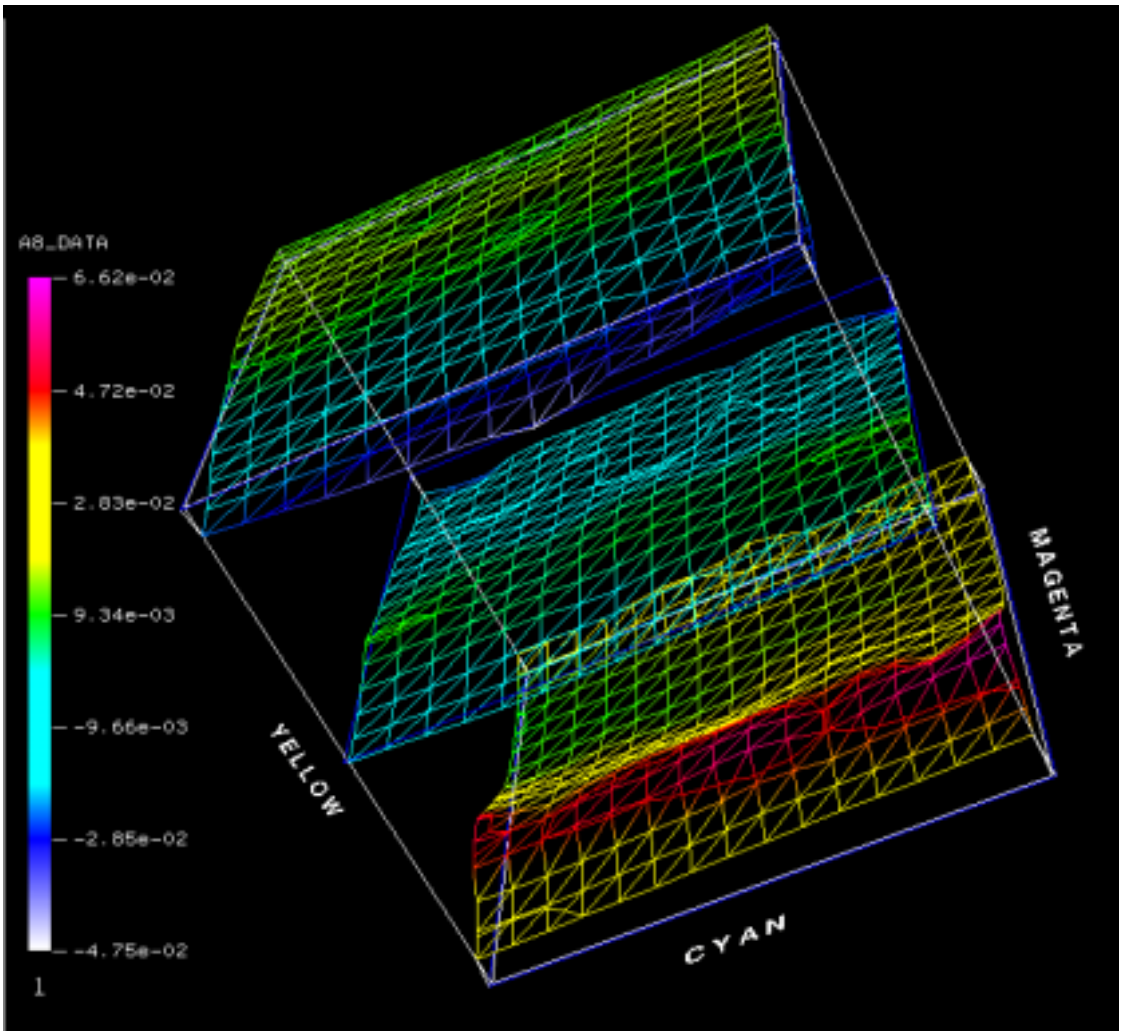

Figure 5.3: Cut planes for the α^8 coefficient.

depending on the granularity desired by an investigator. This means that a true surface can display the approximation without implying the existence of unknown values.

In displaying each result component individually an investigator can interactively view the basic shape of the fitting hyper-spline. At this point choices made for the spline basis order and number of segments can be evaluated. Should the degree of the basis be raised? This would provide an closer approximation to the data, while incurring additional cost during evaluation. Are more segments needed? This alteration involves a trade-offs between a higher residual and the worry of fitting the noise in the data.

Considering the smoothness of the data, i.e. there were no folds or sharp peaks, initial fits were computed using a two segment spline in each data dimension. Figures, 5.4 - 5.6, show these first approximations of the data presented in the last section.

5.3 Direct Comparison

Not all the questions posed in the last section can be answered with only the fitting surface visible. A direct comparison of the original data with the surface that approximates it may provide additional insights (Figures, 5.7 - 5.9)

What may appear to be a problem spot for the surface, i.e. a rapid change in shape along one of its variables, may simply be an artifact of a very noisy data point. By showing the two together such an anomaly may be better understood.

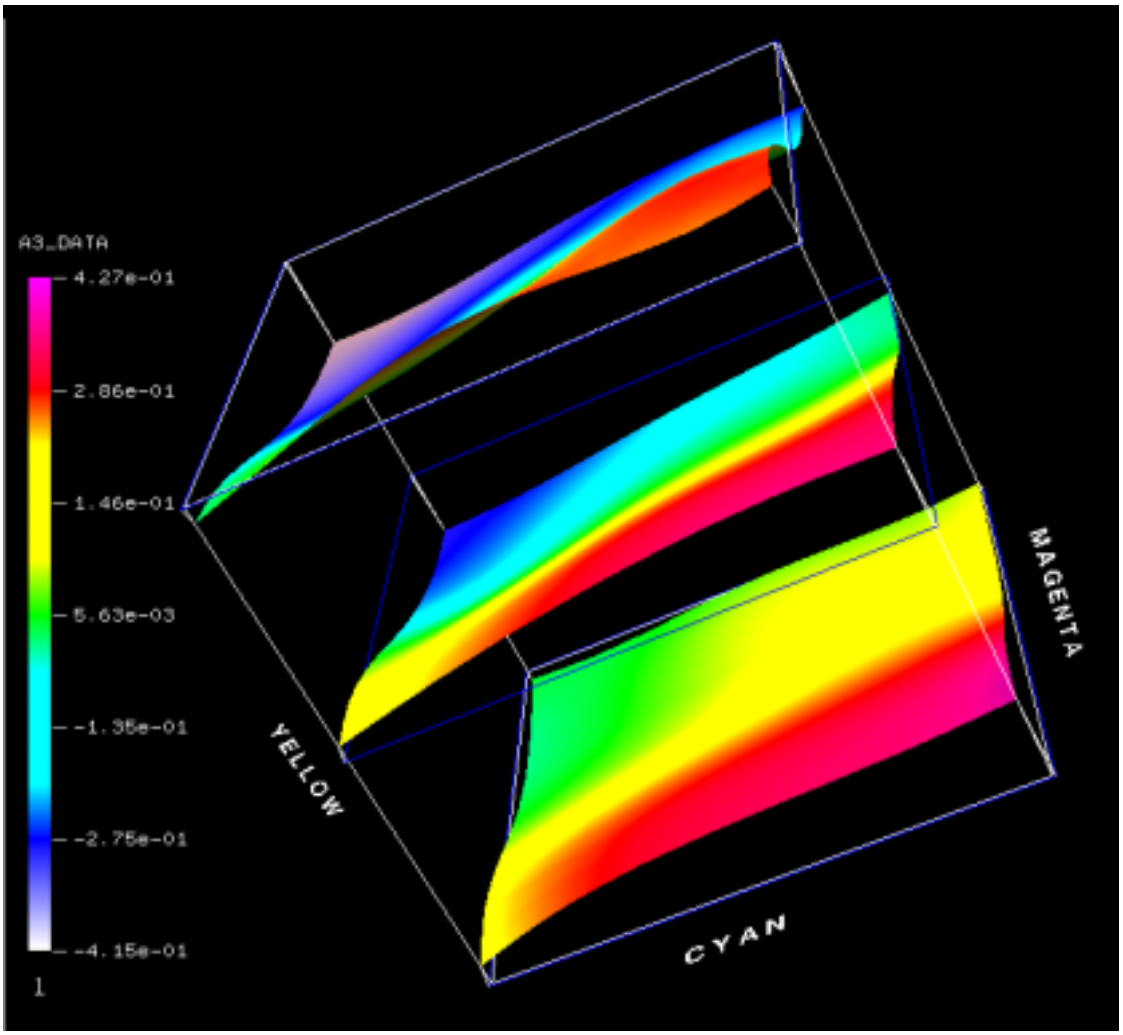

Figure 5.4: Two segment fit of the the α^3 coefficient.

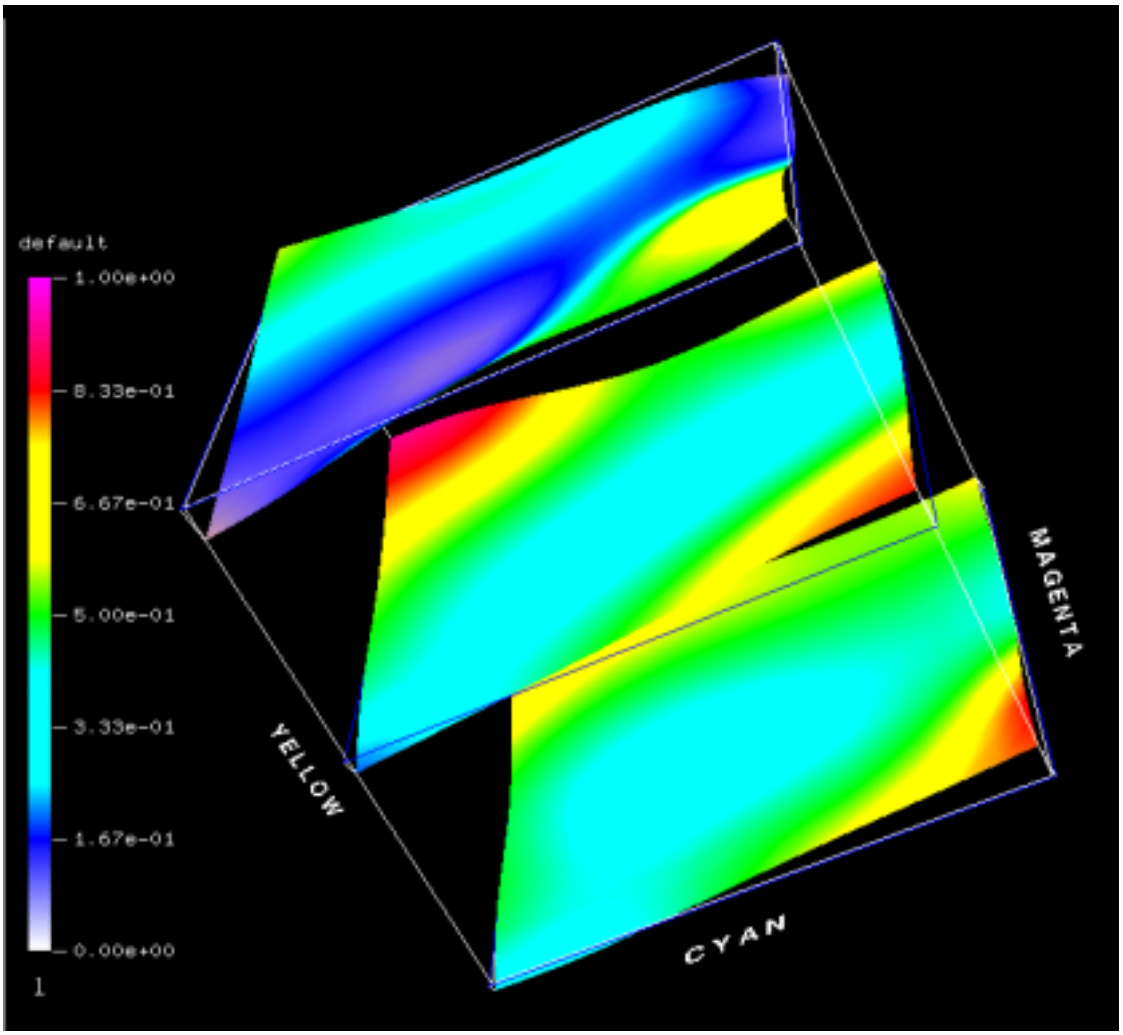

Figure 5.5: Two segment fit of the the α^7 coefficient.

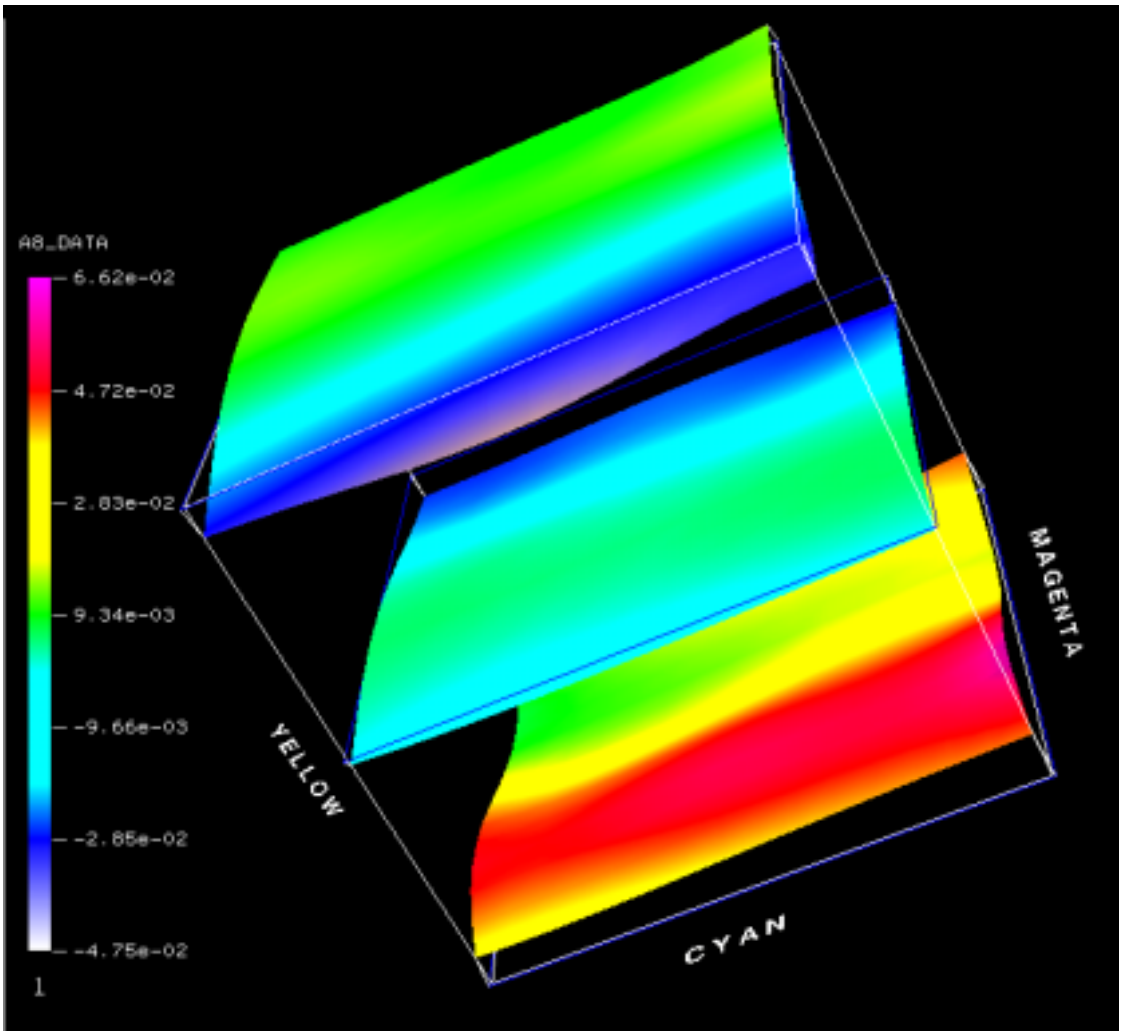

Figure 5.6: Two segment fit of the the α^8 coefficient.

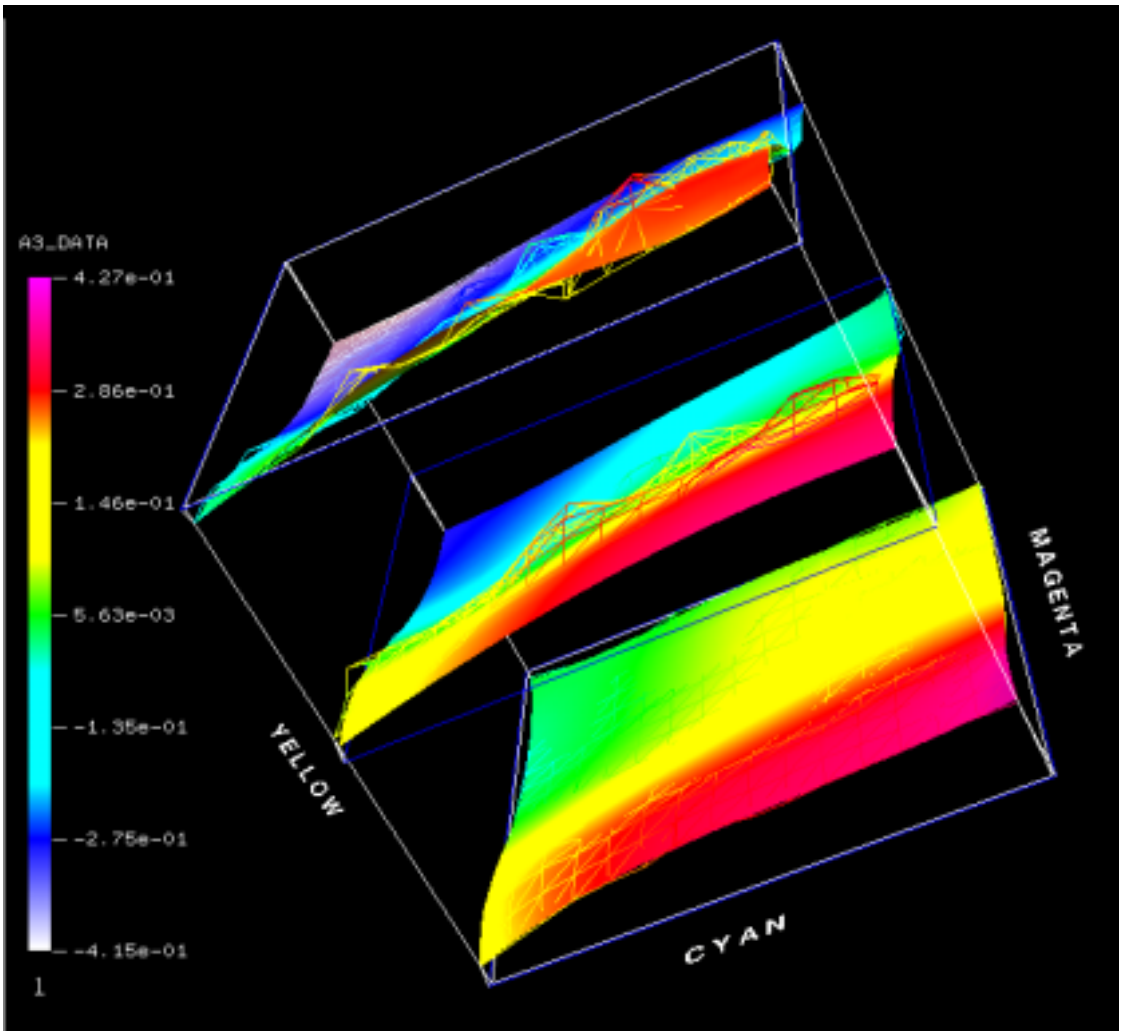

Figure 5.7: Cut planes and fitted surface for the α^3 coefficient.

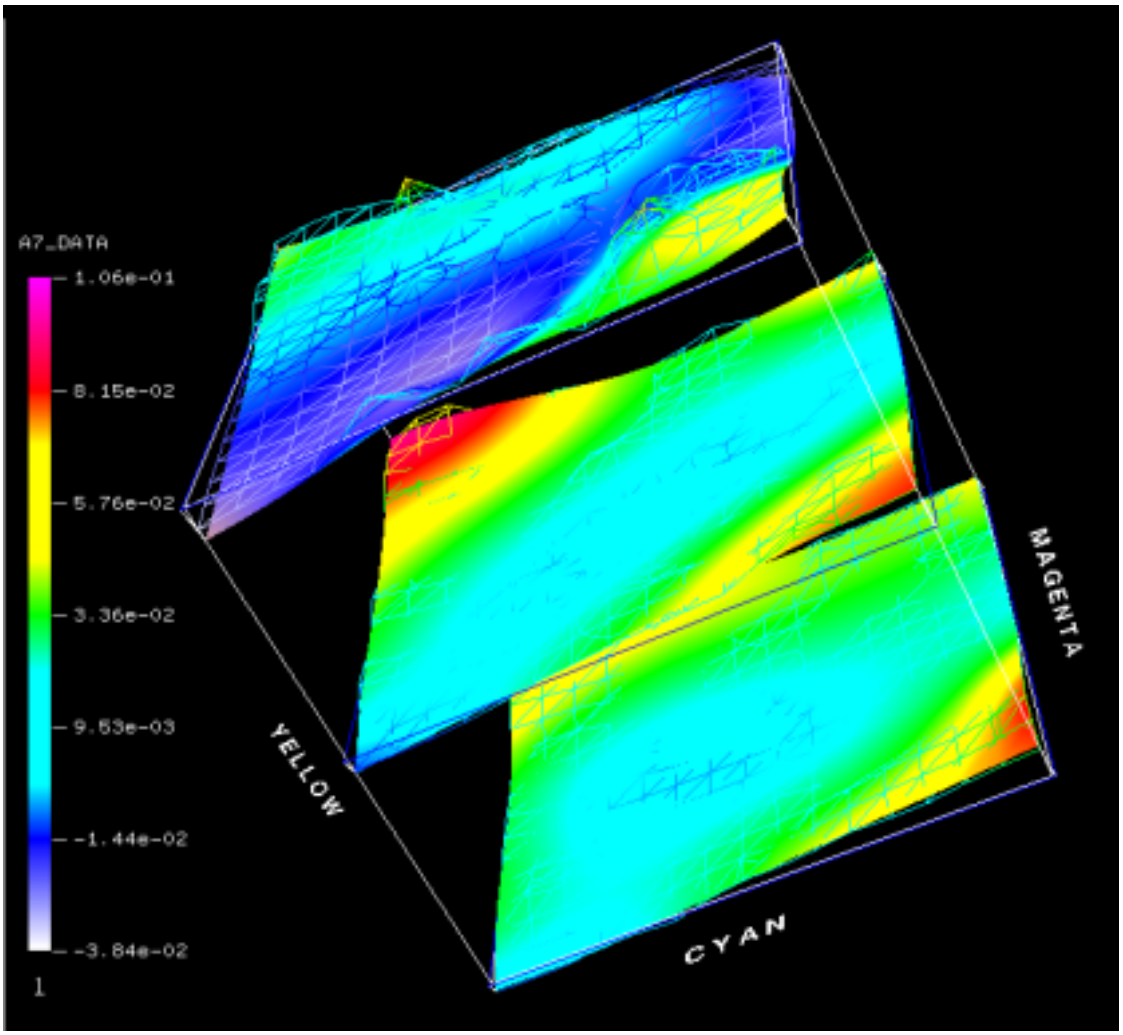

Figure 5.8: Cut planes and fitted surface for the α^7 coefficient.

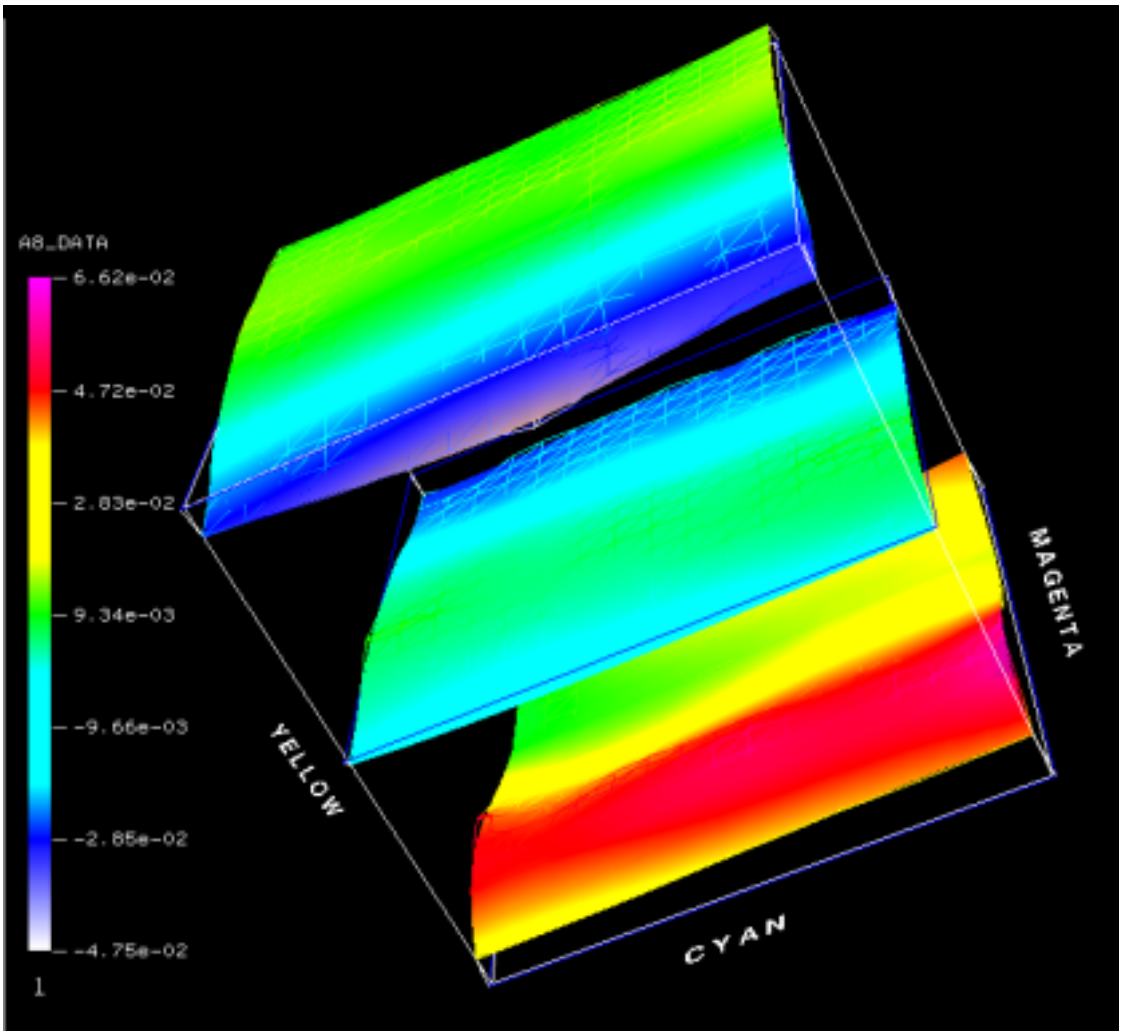

Figure 5.9: Cut planes and fitted surface for the α^8 coefficient.

5.4 Residual Display

To gain a better understanding of the elements that sum to provide the residual value, a new surface may be produced from the difference between the data set and the fitting spline. Of course, the definition of this surface can only be as fine as the sampling points used to compute the residual.

Figure 5.10 outlines the varying levels of success achieved in the initial fit. This provides a simple means to determine if the characteristics of the fitting splines need to be changed.

Working with the assumption that more segments can be introduced without approximating any of the error in the data, a fitting spline was produced with three segments. When this is done it may be difficult to determine the improvement in the approximation. A simple way to analyzing the effects of this change it to visualize the differences between the two residual surface, as is done in Figure 5.11.

For the most part the addition of another segment has brought the fitting surface only slightly closer to the data. The one exception is the data spike in the top slice, with the surface making significant movement towards it.

For this data set the move to splines with four segments would not provide better approximations, due to the influence of noise. At this point the fitting is considered complete. Queries of the hyper-surface will provide access to information needed to perform accurate colour reproduction, as discussed in detail in Chapter 6.

The goal throughout this visualization process is to provide an investigator with intuitive, meaningful, representation of the approximation. While it is true that other methods exist to convey this information, concern here is not with the manner in which the message is delivered, only that details required to define a suitable hyper-surface are accessible.

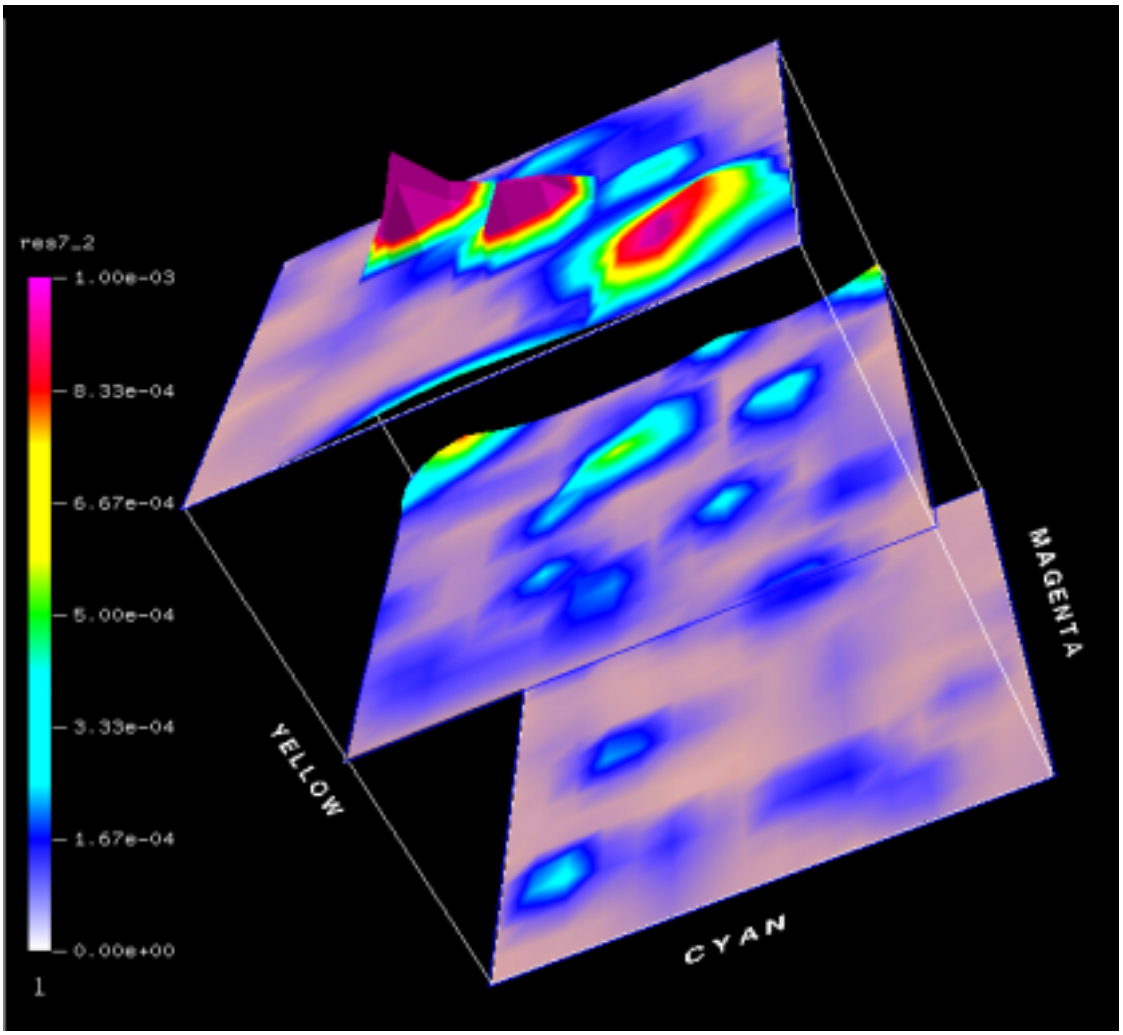

Figure 5.10: Residual values for a two segment fit of the α^3 coefficient.

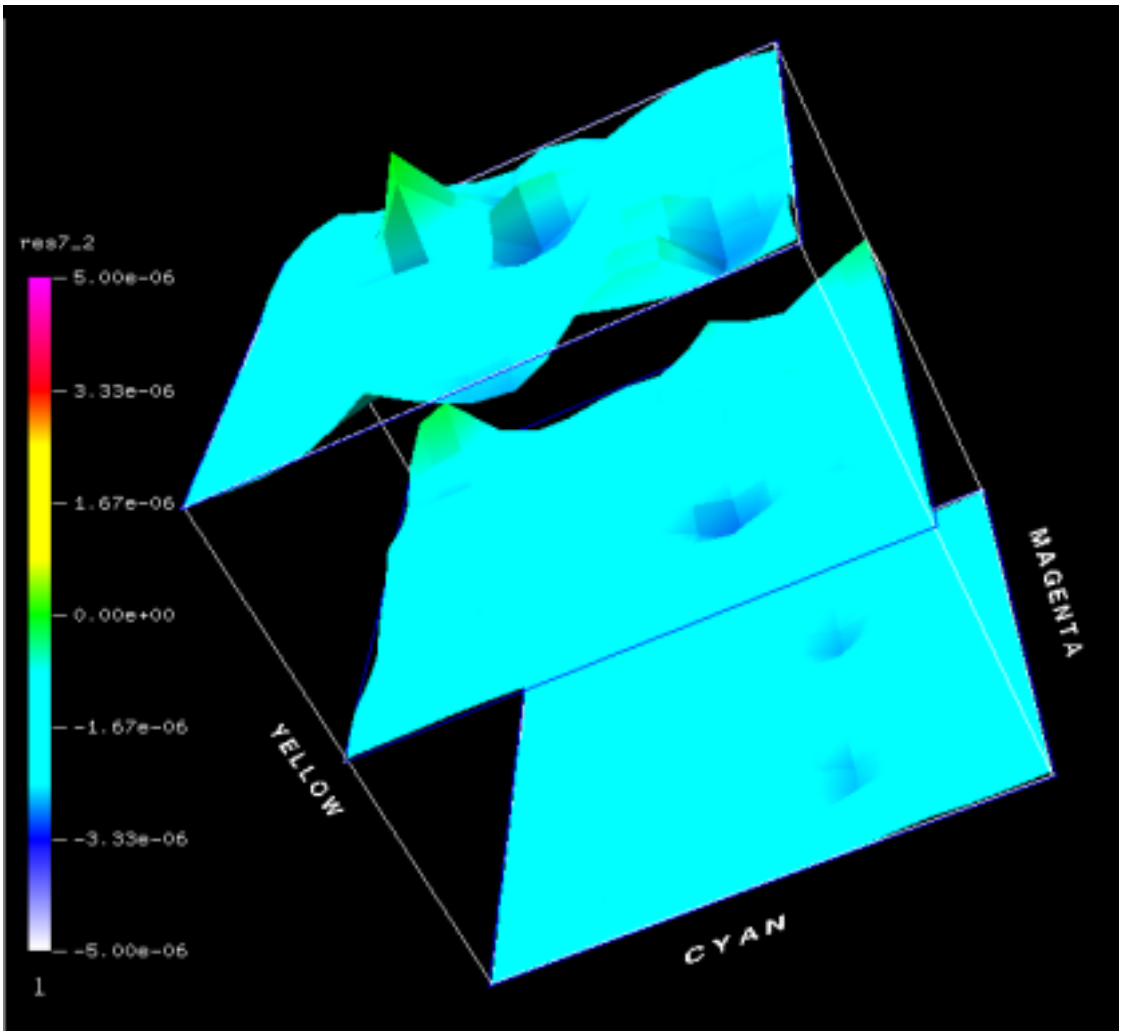

Figure 5.11: Change in residual values after the addition of a third segment to the α^3 fit.

Chapter 6

The Minimization Process

The colour printing hardware used for this work provided data in which the sampling set consisted of a regularly spaced lattice of points in CMYK space. These sets were measured by spectroradiometry, as 8-tuples, $\alpha_1, \cdots, \alpha_8$, of α-basis coefficients. Completion of the fitting process results in a spline hyper-surface that functionally approximates the output characteristics of a specific printer.

The fitting spline, $C(u_1, \ldots, u_n)$, provides a function that maps points in CMYK space to points in the space of the α-bases. This function can be evaluated with tools detailed in Chapter 7. At this point, the following question can be answered: Given a requested colour P_{cmyk}, which is required to be a point in the domain of the hyper-surface, what colour will actually be produced by the printer, $P_{\alpha_1,\cdots,\alpha_8}$? Here the printed colour is a point in the range of the spline

$$C(P_{cmyk}) = C(c, m, y, k) \mapsto \alpha_{1_{cmyk}}, \ldots, \alpha_{8_{cmyk}}.$$

Directly, such information is of little use and poses a demanding problem. When an artist or designer wants a specific colour on a piece of paper they are not concerned

with the inputs provided to the printer. A specific result is desired without concern for the route taken to achieve it.

What is needed is the ability to ask for a desired colour, $P_{\alpha_1,\cdots,\alpha_8}$, and be provided with the inputs that must be sent to the printer, P_{cmyk}, to achieve this result. In terms of a mathematical function, it is the inverse of C that is needed to allow access to the desired colours:

$$C^{-1}(\alpha_{1_{cmyk}}, \ldots, \alpha_{8_{cmyk}}) = P_{cmyk}$$

Since there is no known inverse function for an arbitrary B-spline hyper-surface, an appeal must be made to a minimization process to find particular values of C^{-1}.

The minimization process solves two similar but notably distinct problems. Those are: one, finding the parameter points from the domain of the B-spline, required to reproduce a colour and two, determining a suitable substitute for colours that cannot be produced on a specific printer. The similarity in these problems is due to the fundamental underlying question; what parameter point is closest to the one that produces the desired colour? Even when asked to produce an attainable colour only the closest point approximation will be attained, since the fitting spline itself does not interpolate the data points.

6.1 The Minimization Engine

At the lowest algorithmic level, the function minimization is performed by existing routines [10]. Details of how this was incorporated into these tools can be found in Chapter 7. This is a local minimizer, meaning that it will find the parameter value that minimizes the objective function only if it is provided with a suitable starting point. A complete treatment of such minimization problems is given in [24].

The availability of this minimizer reduces the problem to one of finding a suitable initial guess for the closest point.

Definition 6.1 *The* **minimization problem** *for a point, $P_{i_1,\ldots,i_n}$ consists of finding an initial parameter point $(u_1,\ldots,u_n)$ that permits a minimizing algorithm to converge upon the solution point $(u'_1,\ldots,u'_n)$. Additionally, this convergence must be to a point that minimizes the function*

$$\|C(u'_1,\ldots,u'_n) - P_{i_1,\ldots,i_n}\|.$$

6.2 Conversion to Bézier form

During the fitting procedure any change to control vertices of a Bézier spline would require constraints to preserve continuity, such constraints are not required for B-splines. Now that the fitting process is complete, the control vertices are set and no further changes are required. The derivative conditions outlined in that section are maintained as the resulting B-spline hyper-surface is re-represented as equivalent Bézier splines. This permits the easy computations of first derivative roots.

Given that Bézier curves are a special case of the more general B-spline representation, the conversion of the fitting spline to Bézier form is relatively simple.

Theorem 6.1 *Any NUB-spline whose knots all have order-multiplicity is equivalent to a Bézier that has those points as its breakpoints.*

The conversion process is simply a matter of inserting any necessary knots to raise all knot multiplicities to a sufficient level. The proof of this theorem can be found in [14].

6.3 Seeding the Minimizer

The minimization engine used in this work [10] implements a variation of the secant method [7]. Successful convergence to minima is assured, subject to the criteria that the interval in which the minimization takes place can have, at most, a single change in sign of the first derivative.

Complying with the restriction involves finding the inflection points of the fitting spline and subdividing the Bézier hyper-spline into intervals with the necessary attributes. This was done by subdividing the spline until every segment in each of the n variable dimensions was either strictly increasing or decreasing. There is no theoretical proof that this subdivision has been sufficient to guarantee that a global minimum will be found. The results obtained and presented in Chapter 5 demonstrate that typical surfaces provide initial parameter points in the correct segment, allowing the minimizer to be successful in finding the true minimum. With the hyper-surface sufficiently subdivided, there are two distinct approaches to determine the closest point solution.

6.3.1 The Minimization Method

The process begins by converting the NUB-spline formulation (computed through the stepwise process in Chapter 4) into the equivalent Bézier representation. This conversion is the basis for the refinement criteria detailed in Section 3.1. The conversion is simple, with the multiplicity, $q_{i,j}$, of each knot in each of the vectors, $\{\mu\}_{i=0}^{n}$, being raised to the order of the spline in that dimension, r_i.

The elimination of segments that will trap the local minimizer is a two step process. First, the 'inflection points' of the Bézier hyper-surface are located. This

requires that the polynomial representation of the hyper-surface along each of the n variables be found. From this, zero values in the first derivative are computed. The parameter points of these zeros are used to subdivide the hyper-spline, further demonstrating the need for a spline that can be refined easily. When all the refinements are completed no segment of the hyper-spline remain that might trap the minimizer.

The *control point method* of minimization assumes that the convex hull property of Bézier formulations, along with the subdivision, have made the control points a good approximation of the actual spline. By determining the control point that is closest to the exterior point, a parameter point in the region of influence of that control point can be used as the initial seed to the minimizer. This parameter point is computed via an extension to the Greville point, whose original definition for the B-spline $B_i^r(u)$ is the average of the parameter points $\mu_{i+1}, \ldots, \mu_{i+r-1}$. The extension consists of averaging all the Greville points for the B-splines $B_i^r(u), \ldots, B_{i+r}^r(u)$ influenced by the control point ν_i.

The availability of these initial parameter methods means that a user need only supply the desired external point. An initial parameter will be computed automatically and the minimum found. Details of this procedure follow in Chapter 7 as part of the implementation specification.

This is the conclusive product of work. Current research [5] makes use of these minimization results to demonstrate the feasibility of various cross-rendering methods. The goal here is not to find the definitive solution to this dual media problem. Rather, success is measured in terms of the facility provided by these tools to others.

Chapter 7

Implementation

Previous chapters have described data sets that, by virtue of their rectangular topology, are candidates for fitting. These sets have data dimensions that range from three, for simple CMY observations, to eight, using the alpha bases to measure results. It would be impractical to write an application suitable for these and all other potential data dimensions.

The goal of this work is to provide C++ libraries that can be used to build prototype applications to solve colour fitting problems. Development of these classes was accelerated by building upon existing work [32, 29, 30]. The encapsulation and inheritance features of the C++ language [13] minimized the integration problems typically encountered outside the object oriented paradigm [25]. Steadily increasing the functionality of prototyping tools has been the ongoing goal of the Splines Project since 1989 [35]. This is the fourth work produced as part of this ongoing project. As such it uses existing classes, written by peers, while building new classes that future researchers may use.

While contributing to tools that are of general use to computer graphics re-

searchers, each project within the group produces classes tailored to focused problems. In this case, the tools will help build applications to fit printer spectra data, where the data dimension is arbitrary. Applications will produce hyper-spline formulations that can be used to determine appropriate input values, typically CMYK, to send to a printer to render the desired colour (as measured by the alpha bases) onto paper.

Before the application is outlined, the various support classes must be outlined. There are two categories of classes: those that extend current fixed dimension approximation tools and those that were built to support the unique characteristics of the colour fitting problem. Complete interface details, in the form of manual pages, are provided in Appendix B.

The Splines Project is organized into a series of libraries, each containing a set of closely related or directly inherited classes. The SmallTalk and NIH [26] model, which consists of one large inheritance tree, is avoided in lieu of a forest of many small trees.

7.1 Indexing Schemes

When working with data of arbitrary dimensions the simple indexing schemes normally used to access vector and matrix elements become cumbersome to implement in each of the myriad of potential data dimensions. The `Lattice` class library contains the lowest level of support for multidimensional data by bundling a variety of indexing schemes. Each class in the `Lattice` library implements operators that take an arbitrary number of index values and convert them into a single index, suitable for accessing data stored as a vector.

Storage reduction is provided by the `CompMatIndex` class, which is built from the column tops and extents of the B-spline evaluation matrix, described in Section 3.4. This allows the banded matrix to be indexed as though it were a complete matrix. Avoiding undefined matrix elements is the responsibility of the application, though the class enforces rules to deny such access to non-compliant applications.

Code Sample 7.1 *Constructing an instance of the* `CompMatIndex` *requires the column tops and the number of elements in each column. The* `IntVec` *[29] class is a simple integer array.*

```
IntVec tops(3);                              \\ 3 column matrix
IntVec sizes(3);
tops[0]  = 2; tops[1]  = 2; tops[2]  = 3;    \\ First non-zero row
sizes[0] = 2; sizes[1] = 3; sizes[2] = 5;    \\ Last non-zero row
CompMatIndex ti(tops, sizes);                \\ The indexer
```

Data sets based on a rectangular lattice, with all entries defined, are supported by the abstract base class `Lattice`. All descendant classes, such as `STensorIndex`, must provide an implementation of the multiple to single indexing operator, which is a pure virtual operator in `Lattice`. Structures such as the control net of the hyper-spline and the original data set itself are accessed easily by `STensorIndex`. The intermediate control nets produced during the stepwise fitting process also make use of this indexing scheme.

Code Sample 7.2 *Instances of the* `TensorIndex` *only require the size of each dimension. This example constructs an indexer to access the data from Example 2.2.*

```
IntVec sizes(3);                             \\ 3-d matrix
sizes[0] = 4; sizes[1] = 6; sizes[2] = 5;    \\ of size 4x6x5

TensorIndex ti(sizes);                       \\ The indexer
```

The repeated slicing of data sets required during the fitting process is facilitated by the `STensorIndexPerm` class. The similarity in naming is meant to help users know that this indexes into the same rectangular structure as the `STensorIndex`. The difference is in the interpretation of the index values during an iteration through the index. The order of significant indices within a `STensorIndexPerm` is arbitrary, though fixed at construction time. This user specified ordering makes the determination of slice sets trivial in any data dimension.

Code Sample 7.3 *The slice set from Example 2.6 is easily obtained from the following instance of* `TensorIndexPerm`.

```
IntVec          sizes(3);                   \\ 3-d matrix
IntVec          order(3);
sizes[0] = 4; sizes[1] = 6; sizes[2] = 5; \\ Size 4x6x5
order[0] = 1; order[1] = 0; order[2] = 2; \\ Digit significance

TensorIndexPerm ti(sizes,order);            \\ The indexer
```

Index ranges are not always zero based. When mass evaluations are performed across the domain of a spline, they typically iterate along the entire fully defined range in each dimension. Lattices that are based at non-zero indices are handled by the doubly bounded indexing class `DBTensorIndex`.

Code Sample 7.4 *The size of the* `DBTensorIndex` *in each dimension is implied by the extents along those dimensions.*

```
IntVec bot(3);                              \\ 3-d matrix
IntVec top(3);
bot[0] = 2; bot[1] = 3; bot[2] = 5;         \\ Range minima
top[0] = 3; top[1] = 5; top[2] = 6;         \\ Range maxima

DBTensorIndex ti(bot, top);                 \\ The indexer
```

These indexing methods are rarely used directly. Rather they are used with a data vector, as described in the next section

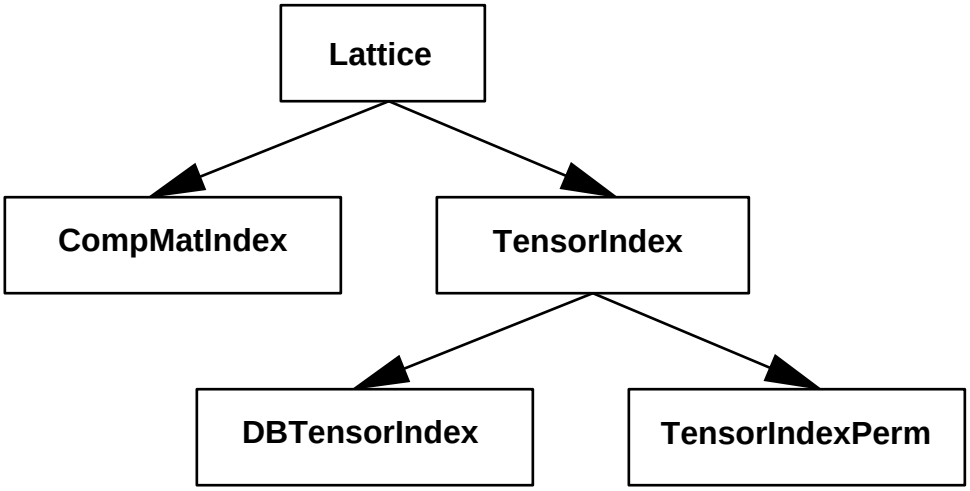

Figure 7.1: Indexing Classes

7.2 Multidimensional Arrays

The power of the indexing classes is seldom used without an associated set of data points. This fact lead to the development of a parallel set of classes to act as data maintainers for the `Lattice` library. These data maintenance classes form the `SIndexedVec` library. Applications and other class member functions rely on classes from this library, rarely accessing the `Lattice` library directly.

The `SCompMat` class is comprised of the indexing class `CompMatIndex` and a vector of floating point values. There is no need to maintain a vector with arbitrary length tuples, as this class is used to store B-spline evaluations. Such evaluations are floating point values, regardless of the number of variables in the approximating hyper-spline.

Code Sample 7.5 *Building a* `SCompMat` *is a simple process, once the underlying indexing class is constructed.*

```
IntVec          tops(3);                    // 3 column matrix
IntVec          sizes(3);
tops[0]  = 2; tops[1]  = 2; tops[2]  = 3; // First non-zero rows
sizes[0] = 2; sizes[1] = 3; sizes[2] = 5; // Last non-zero rows

CompMatIndex    ti(tops, sizes);          // The indexer
SCompMat        mat(ti);                  // The data maintainer
```

The basic indexing functions provided by the `Lattice` class are combined with a vector of arbitrary length tuples to form the abstract base class `SIndexedVec`. This class underlies all maintainers of gridded data that have entries with an arbitrary number of components.

As expected, the `STensorVec` class facilitates access into a rectangular grid of tuples, its indexing operators simply appealing to the corresponding `STensorIndex` member. A similar wrapper was envisioned for all the indexing classes, providing a parallel library.

Code Sample 7.6 *When constructing a* `STensorIndex` *for the control net of a hyper-spline its tuple length must be* $n + t$.

```
IntVec        sizes(3);                          // For a 3-d index
sizes[0] = 3; sizes[1] = 3; sizes[2] = 3;        // of size 3x3x3

TensorIndex idx(sizes);                          // The indexer
STensorVec tup(3, idx);                          // Tuple length is 3
```

The Spline Project is maturing in concert with the C++ language. During the development of the `SIndexedVec` library *templating* [34] C++ compilers became

available. This advance provides means to implement the class `SIdxVec`, which takes classes inheriting from `Lattice` as arguments at instantiation. Data maintenance for the `STensorVecPerm` indexer is provided by `SIdxVec`. Other classes, such as `SCompMat` and `STensorVec`, were rendered redundant by the arrival of templating compilers. They remain in the delivered tools as time constraints did not permit complete retrofitting of existing code.

Code Sample 7.7 *Many types of data maintainers can be created in a similar fashion with the availability of templates. The indexer type must be supplied along with the data storage type and the tuple length. Complete details of the appropriate data types are in the Appendix B.*

```
IntVec bot(3);                              \\ 3-d matrix
IntVec top(3);
bot[0] = 2; bot[1] = 3; bot[2] = 5;         \\ Range minima
top[0] = 3; top[1] = 5; top[2] = 6;         \\ Range maxima

DBTensorIndex ti(top, bot);                 \\ The indexer

SIdxTuple<DBTensorIndex, DoubleVec, TupleVec> tup(idx, 4);
```

Code Sample 7.8 *The flexibility of the* `SIdxTuple` *is shown here as another type of indexer is used to access data.*

```
IntVec sizes(3);                            \\ 3-d matrix
IntVec order(3);
sizes[0] = 4; sizes[1] = 6; sizes[2] = 5; \\ Size 4x6x5
order[0] = 1; order[1] = 0; order[2] = 2; \\ Digit significance

TensorIndexPerm ti(sizes,order);            \\ The indexer

SIdxTuple<TensorIndexPerm, DoubleVec, TupleVec> tup(idx, 4);
```

It is important to note that instances of both `STensorVec` and the `STensorIndexPerm` version of `SIdxVec` can share the same set of data, since the data is always maintained as a simple vector. This provides an easy means to manipulate the same data from varying perspectives.

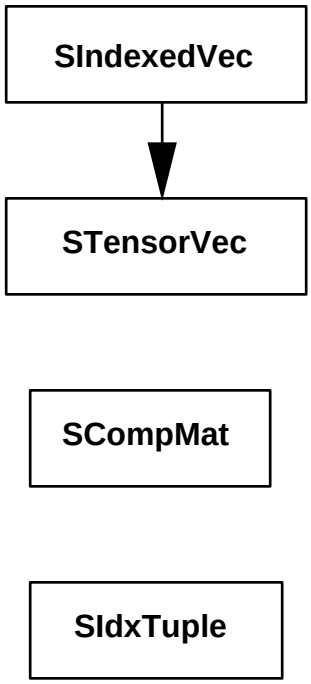

Figure 7.2: Data Classes

7.3 Multivariable Splines

The support for multidimensional data sets provided by the `SIndexedVec` library minimizes the work required to implement classes for splines with an arbitrary number of variables. These classes form the `SHypSurf` library.

Existing B-spline basis classes provide tested evaluation routines. Their correctness is assumed, given the history of their use in the development of B-spline

curve and surface classes [32].

The hyper-splines required for this work are tensor products. Extending existing implementations in two variables to functions in n variables was straightforward. While many evaluation member functions exist (see Appendix B) they all center on the single point evaluation routine that implements the B-spline bases, control vertex formula from equation (4.1). The new classes `SHypSpline` and `SHypBSpline` abstract the notions of a tensor product surface and a tensor product B-spline surface respectively. This abstraction accelerates the implementation of two restricted forms of B-spline hyper-surfaces, namely the `SHypNUBSpline` and `SHypBezSpline`. These are used to maintain nonuniform B-spline hyper-surfaces (produced by the fitting process) and their Bézier counterparts used by the minimizer, respectively.

Code Sample 7.9 *All that is required to create a hyper-spline for fitting are the orders and dimensions. For brevity the knot vectors are initialized to their defaults,* $0, 1, ..., r_i + m_i$. *Only the number of components are required for the control net, as the structure of the lattice is implied by the bases. The entries of the control net are computed during the fitting process.*

```
IntVec ords(3,4);                 // All orders are 4
IntVec dims(3,4);                 // All dimensions are 4

SHypNUBSurf t(ords, dims, ns, 8);   // 3-variable spline with
                                  // 8-d control points, knot
                                  // vectors default to 1,2,...,8
```

Code Sample 7.10 *The Bézier counterpart can be constructed from the same information as the nonuniform spline. Alternatively, as is done in the prototype applications a* `SHypNUBSurf` *can be used as the construction parameter.*

```
IntVec ords(3,4);
IntVec dims(3,4);

SHypBezSurf s1(ords, dims, 3);  // A Bezier-volume

SHypNUBSurf s2(ords, dims, 3);  // A NUB-volume
SHypBezSurf s3( s2 );           // A Bezier-volume
```

Given the need for hyper-splines whose segments guarantee convergence by the minimizer, the Bézier representation may require subdivision. The points at which subdivision is to occur are found by the class `BezCV2Poly`, which provides a polynomial representation of the Bézier hyper-surface. Such a formulation can be passed off to existing polynomial classes for inflection point location.

Code Sample 7.11 *The process of finding first derivative roots is trivializes by the* `BezCV2Poly` *class. Here, the polynomial representation of a Bézier spline is found via a member function.*

```
BezCV2Poly    conv(3);          // Converts a 4th order Bezier
DoubleVec     cvs(5,0,1);       // The cvs {0,...,4}

//
// Compute the polynomial co-effs for the given control net
//
DoubleVec     coeffs = conv.getCoeffs(cvs);
```

7.4 Multidimensional Fitters

Continuing with the extension of existing work to arbitrary dimensions, the `SHypSurfFit` library was implemented. This development completes the progression from existing curve fitters and surface fitters to the approximation of data by hyper-splines.

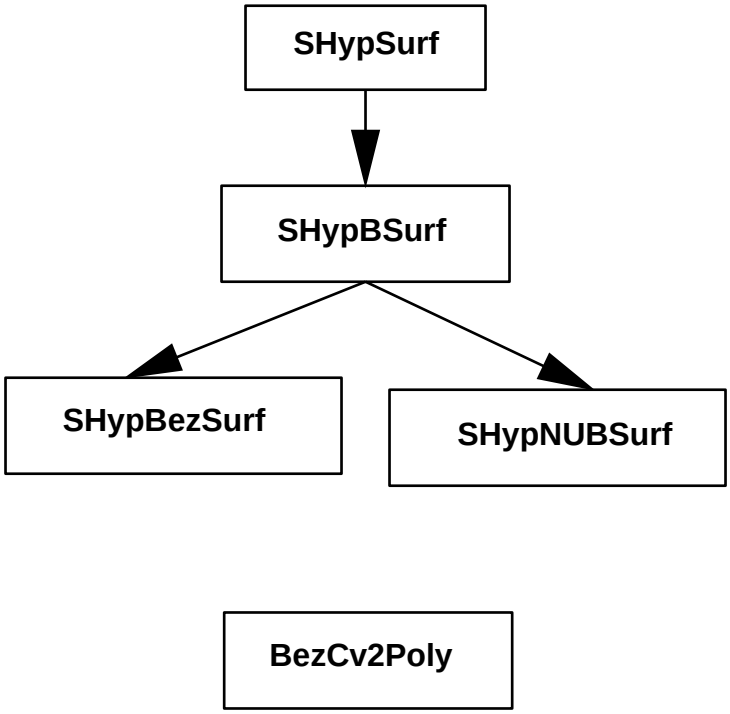

Figure 7.3: Multivariable Spline Classes

Following the same design as its predecessors, the `SHypSurfFit` classes maintains a set of data to be approximated and a B-spline formulation. At the outset, this B-spline contains only basic information such as the orders and knot vectors for each variable. The fitter completes the structure by computing the control vertices based on the data set. While the curve and surface fitters use the general facilities of the LINPACK library to derive their control nets, the multidimensional fitter uses the Householder method detailed in Section 3.5.

Code Sample 7.12 *The fitting of a data set with the hyper-spline from Code Sample 7.9 involves just one member function, once the data and surface are combined within the fitter.*

```
SHypNUBSurf surf( ords, dims, tupleLen );
SHypNUBSurfFit fitter(data);

fitter.defaultSurfaceSet( &realSurf );
fitter.lsFit();

cout << "The fitted surface is " << realSurf << "\n";
```

7.5 Linear Algebra Classes

To maximize the advantage provided by the banded format of the B-spline evaluation matrix, a custom library, `SLinAlg`, was written. The `SLeastSqHH` class performs a QR-factorization on a matrix, maintained by the `SCompMat` class, to find a solution to the approximation problem via Householder transformations. Though consisting of a single class, the `SLinAlg` library has been created to highlight the main attribute of its member, the ability to factorize a matrix. Future

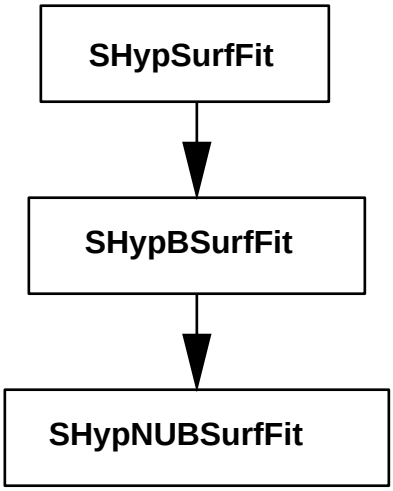

Figure 7.4: Fitting Classes

work with special format matrices may produce classes with functionality similar to `SLeastSqHH`. The extraction of common functionality would permit the development of a base class for generalized factorization methods.

Code Sample 7.13 *Using the indexer from Code Sample 7.5 a* `SCompMat` *is outlined. The entries in the basis evaluation matrix are assumed to be provided from the input stream. Factorization of the supplied matrix is completed as part of the class construction. The* `SLeastSqHH` *is then ready to solve to linear systems, when provided with a data vector.*

```
SCompMat        mat( ti );       // The data maintainer
SLeastSqHH      hh( mat );       // Factorizing class

DoubleVec rhs;
cin << rhs;
cout << "Solution is " << hh.solve( rhs ) << "\n";
```

Figure 7.5: Linear Algebra Classes

7.6 Minimization Classes

The minimization library, `SMinimizer`, provides a C++ interface to a local minimizer [10] along with several initial parameter routines. These combine to provide a global minimizer for descendants of `SHypBSurf`. The success of the minimization process depends on the topological characteristic of the underlying fitting surface. The removal of potential trouble areas, as described in Chapter 6, is essential in allowing this class to converge consistently onto the desired minimum values. This too is a fledgling library and future work may add to its contents. Meanwhile, the unique functionality of its sole class justified the existence of a separate library.

Code Sample 7.14 *The minimization class only requires a surface over which it can compute points. This is used in the search for those parameter values whose evaluation lies closest to the user specified point.*

```
SHypNUBSurf aSurf;                \\ A surface instance
cin >> aSurf;                     \\ Read in details

SHypSurfMinim  mine( &aSurf );    \\ Attach surface to minimizer

parVals = mine.closest( point );\\ Using the minimizer
```

Figure 7.6: Miminimization Classes

7.7 Data Input/Output

All classes produced within the Splines Project provide member functions to support object persistence. This mechanism provides a simple communication path between applications.

In this case the typical split was at the hyper-spline class. One application would read in the data and perform the fitting. The resultant hyper-spline could then be written out to disk, to be used by various applications. Those applications include one that reads in the hyper-spline and formats an output stream readable by the Data Visualizer. This might be termed a format manipulation application, as is commonly found in image processing. A second type of application reads in the hyper-spline to perform minimizations based on user inputs. Typically, the original sampling points, which span the printer gamut, would be provided to such a minimization application to generate the inputs needed to produce the desired colours.

7.8 Application Prototyping

The applications built using these libraries will typically have three distinct components:

- Colour data fitting

- Closest point minimization
- Post-processing

While initial experiments are being conducted, only the fitting component is required. Once the user is satisfied with the resulting fits, the application can be extended to include the minimizer. Alternatively, two smaller applications can be written, with the necessary data transferred between them using the persistence operators. The minimizer permits interactive querying of the fitted surface to determine closest point solutions. The results of such queries can be used to determine the input values that must be sent to the printer to get the closest possible approximation to the desired colour, irrespective of the desired colour's location with respect to the printer's gamut.

Finally, post-processing is meant to include any analysis that is done on the minimization results.

7.8.1 Data Fitting

Data Input If the data is provided in a format that is understood by the input operators of the `STensorVec`, this step reduces to calling that member function. Otherwise, the data must be converted into such a format.

Spline Characteristics As outlined in Section 6.3 details of the fitting hyperspline must be provided to seed the process. The orders and knot vectors, which imply the spline space dimension, must be fed to the application.

Build Hyper-Surface With the essential characteristics of the fitting spline available the application can build an instance of the `SHypNUBSurf` class. For now the control net remains undefined.

Build Fitter The data and the hyper-spline are brought together at this point to create an instance of the `SHypNUBSurfFit` class. The entire fitting process is contained within its member functions. This allows the approximation to be easily performed by calling the `SHypNUBSurfFit::lsFit()` function.

7.8.2 The Minimization Process

Once a suitable hyper-spline has been determined the minimization engine can be used.

Prepare spline formulation The fitting spline is readily available when the minimization application is combined with a fitting application. If these two applications are separate the spline must be read from the output stream of the fitter.

Refine problem areas Any potential problem areas within the hyper-spline, as described in Section 6.3.1, must be handled at this point. A polynomial representation is extracted from the Bézier version of the fitting spline by using the `BezCV2Poly` class. Existing derivative and root finding routines provide those points at which the hyper-surface should be refined. This subdivision process is handled by Splines Project classes predating this work.

Build a minimizer This step merely attaches the refined surface to the minimization engine. As the underlying minimizer is a local minimizer, the determi-

nation of a suitable initial parameter is essential. Such a parameter allows the minimizer to converge on the surface point closest to the user supplied external point.

Query minimizer The member functions of the `SMinimizer` class will provide closest point solutions based on the attached surface. The user simply specifies the external point (desired colour) and the initial parameter method. The options for the initial parameter method are detailed in the manual page as well as in Section 6.3. The minimizer will then return a point within the printer's input domain to produce the desired colour.

7.8.3 Post-Processing

With the closest point returned (based on the user's desired colour) the application is now able to interpret the result. This interpretation might be based on a series of queries, i.e. are all the returned colours within some tolerance? The results might be used directly to control a printer, thus providing an accurate reproduction of the desired colour.

As mentioned in the introduction, the mandate of this work is not to provide uses for these results, only to make the results easily accessible to other researchers.

Chapter 8

Future Work

Having achieved the original goal of developing means to fit multidimensional printer data, it is important to consider future directions. Possible paths have become apparent as results from this approach were examined.

Chapter 7 outlines how intervention is required to supply the optimal number of fitting segments. While such information is never simple to determine, the characteristics of printer data may prove helpful. One potential avenue is the development of a statistical method for determining the number of segments that best fit a given data set.

The methods used in Chapter 6 to refine the fitting surface have performed as required. Experience has demonstrated that, in practice, excessive subdivision is occurring. A more rigorous method may be required to approach the optimal refinement of the fitting surface, within the constraints of the minimizer.

A desire to improve upon the existing solutions to these two problems is rooted in the goal of packaging a stand alone application. By polishing existing tools, more convenient, transportable systems can be created. One use of such systems is the

generation of a gamut mapping table to do on-the-fly colour conversion within a digital printer.

All these routes remain faithful to the original assumption that least squares proximity of desired and produced colour is meaningful. Though true, least squares systems are just one of many possible approaches. Currently, the computation of a closest point subject to linear constraints is a desirable goal. Inputs would be the same as those currently used while the system would find the closest point on a result gamut, subject to a given illumination.

The development of new ideas and solutions to them continues. Closing the time gap between these two events is an important task, which was addressed in hopes of releasing the flow of ideas from the worries of implementation.

Appendix A

Summary of Notation

The notation used throughout this work is presented, along with the corresponding page reference of their definitions.

Data Sets

$\alpha^j_{i_1,\ldots,i_n}$	the j-the result component at entry $i_1, \ldots, i_n$ of the data set (9)
δ^i	sampling set in the i-th data dimension, $1 \leq i \leq n$ (9)
θ_i	a slice of the data set in the i-th data dimension (11)
Θ_i	the entire slice set in the i-th data dimension (11)
A	the set result components of a data set (17)
D^n	an n-dimensional data set (10)
$\mathbb{I}_i$	the space of i-tuples (10)
n	the data dimension (9)
t	number of result components in each data set entry (9)

Splines

μ	knot vector of a B-spline (22)
ν_i	element in the control net (24)
ω	intermediate control net, from the stepwise process (56)
$B_i^r(j)$	a i-th B-spline of order r, in the fitting along the j-th data dimension (23)
$C(u)$	B-spline formulation for fitting spline (24, 52)
l_i	index of the interval containing u_i lives (22)
m	dimension of a B-spline, also number of control points (24)
n	number of variables in the spline equivalent to the data set dimension (9)
q	multiplicity of a knot (22)
r	order of a B-spline (23)
V	control net of the fitting spline curve (24)

Matrices

τ	number of parameter points u with $l_{u_i} < i$ (43)
C	matrix of B-spline evaluations (36)
C^i	the i-th column of C (39)
C_i	the i-th row of C (34)
$C^{(i)}$	submatrix of non-zero B-spline values for all u_j where $l_j = i$ (36)
e	column extent of C^1 (43)

Factorization

H_i	the i-th Householder transformation (38)
Q_C	orthogonal component of the QR decomposition of C (20)
R_C	upper triangular component of the QR decomposition of C (20)

Appendix B

C++ Class Manual

In keeping with the goal of providing durable tools for application prototyping, classes within the Splines Project are documented by an automatic manual page generation system. The coordination of code development and documentation ensures that future users of this work are not required to learn complete implementation details.

NAME

Lattice – An abstract class for multi-key to single key indexing

DESCRIPTION

A lattice allows for the creation of a map from a user defined n-dimensional space to a linear indexing scheme. The user provides an outline of the space and the Lattice generates an unique integer for each element in the space.

PREREQUISITES

rwmath, rwtool, STools

SEE ALSO

TriangleIndex, TensorIndex.

AUTHOR

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

COPYRIGHT

CLASS Lattice

Base class(es)
public SError

Friends
ostream& operator<<(
ostream& os,
const Lattice& inst
);

Public members
enum latEnum { BEGIN = -9, END = -99 };

Lattice(void);
A default (no arguments) constructor is included so we can create arrays of Lattice.

Lattice(unsigned n);
A n-dimensional Lattice

Lattice(const Lattice&);
Copy constructor.

virtual ~Lattice(void);
Class destructor.

virtual int operator ()(int);
virtual int operator ()(int, int);
virtual int operator ()(int, int, int);

virtual int operator ()(int, int, int, int);
virtual int operator ()(int, int, int, int, int);
virtual int operator ()(IntVec& i);
The first five opreators simply build an Intvec and call the last version. This exist simply to allow flexible access. They all update the index member.

virtual int operator ()();
Get map of current value in index.

int operator++();
int operator--();
Shorthand for next()/prev().

void deepenShallowCopy(void);
Dealias data.

virtual int getDimension( ) const =0;
virtual void putDimension(unsigned) =0;
Retrieve/change the dimension. The new dimension is returned after the change.

virtual IntVec getIndex(void) const;
virtual int getIndex(int i) const;
virtual int putIndex(int i);
virtual int putIndex(int i, int n);
virtual int putIndex(const IntVec& i);
Retrieve/change the index vector to a specific number or sequence. PutIndex will return the map of the new index.

virtual int prev(void) =0;
virtual int next(void) =0;
Move index to next/previous position and return map of new index.

virtual int first(void) =0;
virtual int last(void) =0;
Move index to next/previous position and return map of new index.

virtual RWBoolean isEOI(void) const ;
virtual RWBoolean isBOI(void) const ;
Check for Begining/End Of Index.

virtual int size(void);
The size of the array indexed by this lattice.

Protected members
IntVec index;
The last index, used for fast repeated access.

virtual RWBoolean verify(void) const =0;
Conditions specific to derived instances of Lattice.

virtual int multiToSingle(void) const =0;
Convert from multiple to single indexes, the multiple index used is the INDEX variable above.

RWBoolean isEqual(const IntVec& iv, int n) const;
Check if all elements of iv are equal to n.

Private members
Nothing

INCLUDED FILES
iostream.h
rw/ivec.h
rw/rwstring.h
STools/SError.h

NAME

CompMatIndex – A compressed matrix lattice

DESCRIPTION

This class converts from multiple to single indices where each multiple index represents a cell in an 2-dimensional virtual matrix. Compressed matrices provide a virtual matrix which allows for traditional indexing operators. The addition of a indexing function allows the data to be stored in the minimum amount of space.

```
IntVec start(4,2), sizes(4,3);
SCompMat(start, sizes);
```

This example will provide a virtual 4 by 6 matrix where only the elements with row indices greater than 1 can be accessed. A graphical output demonstrates the format of the matrix.

```
[ X X X X ]
[ X X X X ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
[ 0 0 0 0 ]
```

In this case a 24 element virtual matrix only requires 15 doubles. Positions marked by an X cannot be accessed. An access error will occur if any non-data element is accessed.

PREREQUISITES

rwmatx, rwtools, STools

SEE ALSO

SIndexedVec

AUTHOR

Bruce Hickey, bhickey@watcgl

BUGS

Only the standard output primitives are supported for input, the graphical representation is not.

COPYRIGHT

CLASS CompMatIndex

Base class(es)
public Lattice

Friends
istream& operator>>(
istream& is,
CompMatIndex& inst
);
ostream& operator<<(

ostream& os,
const CompMatIndex& inst
);

Public members
CompMatIndex(void);
NULL Constructor, for arrays of CompMatIndex.

CompMatIndex(IntVec& v1, IntVec& v2);
A CompMatIndex with row data beginning at v1 and having a length of v2.

CompMatIndex(const CompMatIndex&);
Copy Constructor.

˜CompMatIndex(void);
Class destructor.

void reshape(const IntVec&, const IntVec&);
Change the tops/lenghts of the columns in the lattice.

CompMatIndex& operator = (const CompMatIndex& i);
Assign operations.

virtual int operator ()( );
virtual int operator ()(int);
virtual int operator ()(int, int);
virtual int operator ()(int, int, int);
virtual int operator ()(int, int, int, int);
virtual int operator ()(int, int, int, int, int);
int operator()(IntVec& i);
The indexing routines.

int operator == (const CompMatIndex& i);
int operator != (const CompMatIndex& i);
Index comparison.

virtual int next(void);
virtual int prev(void);
Set to next/previous index from the current index.

virtual int getDimension(void) const ;
virtual void putDimension(unsigned);
Retrieve/change the dimension of the compressed matrix YOU CANNOT CHANGE THE DIMENSION FOR THIS CLASS. The putDimension function exists only for compatability.

const IntVec& getTops(void) const ;
void putTops(IntVec&);
Retrieve/change the first element indices.

const IntVec& getSizes( ) const ;
void putSizes(IntVec&);
Retrieve/change the number of elements in the columns.

int isFirst(void) const ;
int isLast(void) const ;
Check for first/last index.

int first(void);
int last(void);
Set to the first/last index.

virtual RWString className(void) const;
Class id

void printOn(ostream&, RWBoolean gFlag=FALSE) const;
void scanFrom(istream&);
I/O routines.

Private members
IntVec firstElem, colSizes;
DoubleVec vec;
Data vector, and its limits;

virtual int multiToSingle(void) const ;
Converting multiple index to linear index.

virtual RWBoolean verify(void) const;
virtual void checkEnds(void) const;
virtual void checkSizes(void) const;
virtual RWBoolean checkBounds(void) const;

static const unsigned fixedCMIDimension;
For now, we will only deal in 2-D

INCLUDED FILES
rw/ivec.h
rw/dvec.h
Lattice.h

NAME

DBTensorIndex – A rectangular lattice with both upper and lower bounds

DESCRIPTION

A DBTensorIndex provides the same functionality as a TensorIndex with the addition of a lower bound on each index. While the range of a TensorIndex is : [0, uMax[0]-1] ... [0, uMax[last]-1] that of a DBTensorIndex is [uMin[0], uMax[0]] ... [uMin[last], uMax[last]].

```
IntVec top(3), bot(3);
x[0] = 5;   x[1] = 3;   x[2] = 5;
bot[0] = 1; bot[1] = 0; bot[2] = 1;
int const len = 4;
DBTensorIndex ti1(bot, x);
DBTensorIndex ti1(bot, x, DBTensorIndex::TOP);
   // These both index into the array [[1..5],[0..3],[1..5]]
DBTensorIndex ti2(bot, x, DBTensorIndex::LEN);
   //Indexes into the array [[1..2],[0..5],[1..4]]
DBTensorIndex ti3(bot, len);
   //Indexes into the array [[1..4],[0..3],[1..4]]
int  size1 = prod(x-bot+1)    = ti1.size();
int  size2 = prod(x)      = ti2.size();
int  size2 = pow(len,bot.length())  = ti3.size();
<Type> array[t1.size()];  // The data
```

These functions allow you to declare arrays of the sizes given and index into them without keeping track of each bound yourself.

PREREQUISITES

TensorIndex, VecMtx, RWTools, STools

SEE ALSO

TriangleIndex

AUTHOR

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

COPYRIGHT

CLASS DBTensorIndex

Base class(es)
public TensorIndex

Friends
istream& operator>>(
istream& is,
DBTensorIndex& inst
);
ostream& operator<<(

```
        ostream& os,
        DBTensorIndex& inst
);
```

Public members

enum mode { TOP, LEN };
Two different constructor types.

DBTensorIndex(void);
NULL Constructor.

DBTensorIndex(const DBTensorIndex&);
Copy Constructor.

```
DBTensorIndex(
        const IntVec& bot,
        const IntVec& x,
        mode m=TOP
);
```

DBTensorIndex(const IntVec& bot, unsigned len);
Constructor with initialization values.

~DBTensorIndex(void);
Destructor.

DBTensorIndex& operator = (const DBTensorIndex& i);
Assign operator.

int operator == (const DBTensorIndex& i);
int operator != (const DBTensorIndex& i);
Index comparison.

IntVec top(void) const;
IntVec bot(void) const;
unsigned top(unsigned) const;
unsigned bot(unsigned) const;
Retrieve the top/bottom index for all/the specifed dimension.

void top(unsigned i, unsigned n);
void bot(unsigned i, unsigned n);
Change the size along the Nth parameter space.

void reset(void);
Set the top/bottom to zero.

virtual RWString className(void) const;
Class id.

virtual IntVec getIndex(void) const;
virtual int getIndex(int i) const;
Get the current index.

void putDimension(unsigned n);

Change the dimension.

void printOn(ostream&);
void scanFrom(istream&);
I/O routines.

Protected members
IntVec bottom;
First index along each parameter

const static RWString classname;
Class id.

Private members
Nothing

INCLUDED FILES
TensorIndex.h

NAME

SIdxTuple – Vectors of elements indexed by lattices.

DESCRIPTION

This class holds data the can be indexed by a variety of existing lattices. Additionally the data can be of several types. The user must specify the data type and the element type at instantiation. The possible template instantiations are:

```
SIdxTuple< "Lattice", Triple,  TupleVec >
SIdxTuple< "Lattice", SPoint,  SPointVec >
SIdxTuple< "Lattice", SPoint,  SDPointVec >
SIdxTuple< "Lattice", SVector, SVectorVec>
SIdxTuple< "Lattice", SVector, SDVectorVec >
```

where "Lattice" can be any class derived from Lattice

PREREQUISITES

Lattice, Tuple, STools, RWTools, VecMtx

SEE ALSO

STriVec, STensorVec

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

COPYRIGHT

TEMPLATE CLASS SIdxTuple

Template argument(s)
class latticeType
class PointType
class VecType

Base class(es)
public SError

Public members

SIdxTuple(void);

Default constructor.

SIdxTuple(SIdxTuple&);

Copy constructor.

SIdxTuple(latticeType lat, unsigned vecElemLen=3);

Creates a VecType, with tuples of length vecElemLen, the size of the VecType is determined bt the lattice. Since we typically do work in 3-space the element length defaults to this.

˜SIdxTuple(void);
Class destructor.

inline PointType operator()(unsigned a);
inline PointType operator()(unsigned a, unsigned b);
inline PointType operator()(
unsigned a,
unsigned b,
unsigned c
);
inline PointType operator()(
unsigned a,
unsigned b,
unsigned c,
unsigned d
);
inline PointType operator()(IntVec&);
Data access.

inline latticeType& refLattice(void);
Return a reference to the lattice. This is designed to be overloaded be derived classes to return a reference of their type.

virtual inline RWString className(void) const;
Class id.

void scanFrom(istream&);
void printOn(ostream&);
I/O primitives

Protected members
VecType data;
The data is contained in the SIdxTuple.

latticeType lattice;
The indexing function, a copy is made and retained by the SIdxTuple.

void verify();
Input verification.

Private members
Nothing

OPERATORS, FREE FUNCTIONS, CODE, ETC.

template<
class latticeType
class PointType
class VecType
>
istream& operator>>(
istream& is,
SIdxTuple<latticeType, PointType, VecType>&

```
);

template<
        class latticeType
        class PointType
        class VecType
>
ostream& operator<<(
        ostream& os,
        SIdxTuple<latticeType, PointType, VecType>&
);
```

INCLUDED FILES

iostream.h
STools/SError.h
Tuple/TupleVec.h

NAME

SIndexedVec – Base class for Lattice indexed tuple vectors

DESCRIPTION

This is an abstract class for indexed TupleVecs. A derived class must provide an indexing function which is derived from the Lattice class.

PREREQUISITES

Lattice, Tuple, STools, RWTools, VecMtx

SEE ALSO

STriVec, STensorVec

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

COPYRIGHT

CLASS SIndexedVec

Base class(es)
public SError

Public members

SIndexedVec(void);
Default constructor.

SIndexedVec(const SIndexedVec&);
Copy constructor.

SIndexedVec(TupleVec& data, Lattice* x);
Creates a TupleVec, with tuples of length tLen.

SIndexedVec(
unsigned int tLen,
unsigned int size,
Lattice* x
);
Creates a TupleVec, with tuples of length tLen.

~SIndexedVec(void);
Class destructor.

inline DoubleVec operator()(unsigned int) const;
inline DoubleVec operator()
(
unsigned int,
unsigned int
) const;

inline DoubleVec operator()
(
unsigned int,
unsigned int,
unsigned int
) const;
inline DoubleVec operator()
(
unsigned int,
unsigned int,
unsigned int,
unsigned int
) const;
inline DoubleVec operator()(IntVec&) const;
Data access.

inline const Lattice& refLattice(void) const;
Return a reference to the lattice. This is designed to be overloaded be derived classes to return a reference of their type.

inline const TupleVec& refData(void) const;
Return a reference to the data.

virtual RWString className(void) const=0;
Class id.

unsigned int tupleLength(void) const;
The length of the tuples in the vector.

Protected members
TupleVec data;
The data

Lattice *baseIdx;
Pointer to the indexing function.

Private members
Nothing

INCLUDED FILES
iostream.h
STools/SError.h
Lattice/Lattice.h
Tuple/TupleVec.h

NAME

SCompMat – A virtual 2-D matrix

DESCRIPTION

SCompMat(s) gives the illusion of a complete 2-D matrix of doubles but only stores data for valid positions in the indexing lattice. This is essentially a data container for the CompMatIndexer lattice. This class is not derived from the SIdxTuple class because it handles tuples and I only want doubles.

PREREQUISITES

Lattice, Tuple, STools, RWTools, VecMtx

SEE ALSO

CompMatIndexer

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

COPYRIGHT

CLASS SCompMat

Base class(es)
public SError

Friends
ostream& operator<<(ostream&, SCompMat&);
istream& operator>>(istream&, SCompMat&);

Public members

SCompMat(void);
Default constructor.

//SCompMat(SCompMat&); Copy constructor.

SCompMat(CompMatIndex& indexer);
Creates a doubleVec with the size given by the indexer

SCompMat(DoubleGenMat& mat);
Builds an SCompMat from a DGEMatrix-- BE CAREFUL -- this function does a test with zero to determine the bounds of the matrix.

~SCompMat(void);
Class destructor.

operator DoubleGenMat();
Returns an uncompressed matrix with a copy of this data.

void reshape(const IntVec&, const IntVec&);

Reshape the matrix - useful when you've contructed an array of this class and need to add some detail to it.

const CompMatIndex& refLattice(void) const;
Reference to the indexer.

DoubleVec col(int j);
double& operator()(unsigned int, unsigned int) ;
Data access.

unsigned int topPos(unsigned int) const;
unsigned int botPos(unsigned int) const;
unsigned int diagPos(int j) const;
How far into the data-only column is the real diagonal position.

unsigned int rows(void) const;
unsigned int cols(void) const;
Info members

virtual RWString className(void) const;
Class id.

void printOn(ostream&, RWBoolean grFlag=FALSE);
void scanFrom(istream&);

void verify(void);

Protected members
DoubleVec data;
The data

CompMatIndex idx;
Pointer to the indexing function.

Private members
static const RWString classname;
Identifying name for the class.

INCLUDED FILES
iostream.h
rw/dvec.h
rw/dgenmat.h
STools/SError.h
Lattice/CompMatIndex.h

NAME

STensorVec – A multi-dimensional tensor.

DESCRIPTION

This class holds and indexes into a multi-dimensional array, such as those used to maintain the control mesh of a rectangular Bezier volume.

PREREQUISITES

Lattice, Tuple, STools, RWTools, VecMtx

SEE ALSO

TensorIndex, STriVec, SIndexedVec

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

COPYRIGHT

CLASS STensorVec

Base class(es)
public SIndexedVec

Friends
istream& operator>>(istream& is, STensorVec& inst);
ostream& operator<<(
ostream& os,
const STensorVec& inst
);

Public members

STensorVec(void);
The default constructor.

STensorVec(unsigned tLen);
Create a TensorVec with tuples of size tLen, the default Tensor Index will be used. The data can be customized via the reshape() function.

STensorVec(unsigned tLen, TensorIndex& x);
Create a TensorVec with tuples of size tLen using x's indexing scheme.

STensorVec(TupleVec& theData, TensorIndex& x);
Use the given data and x's indexing scheme.

STensorVec(const STensorVec& x);
Copy constructor.

~STensorVec(void);

Class destructor.

void printOn(ostream&) const;
void scanFrom(istream&);
I/O routines.

virtual inline RWString className(void) const;
Class id.

STensorVec& operator=(const STensorVec&);
Assignement operator - copies data.

STensorVec& operator+=(const STensorVec&);
STensorVec& operator-=(const STensorVec&);
STensorVec& operator*=(const STensorVec&);
STensorVec& operator/=(const STensorVec&);
Vector arithmetic.

STensorVec& operator+=(double);
STensorVec& operator-=(double);
STensorVec& operator*=(double);
STensorVec& operator/=(double);
Scalar arithmetic.

void deepenShallowCopy(void);
Used to dealias data - copy constructors alias data.

STensorVec copy(void) const;
Make a separate copy.

inline const TensorIndex& refLattice(void) const;
Reference to the indexer.

inline unsigned size(unsigned n) const;
Get the size of the TensorVec in the Nth dimension.

void resize(unsigned dim, unsigned sz);
Resize the the TensorVec in the Nth dimension, data may be corrupted.

unsigned dimension(void) const;
void dimension(unsigned);
Query/change the dimension of *this.

void reshape(const IntVec&);
void reshape(unsigned, const IntVec&);
Change the shape of the underlying lattice as well as resizing the data vector. The second case allows you to resize the tupleLength of the data.

void trim (const IntVec& start, const IntVec& len);
Trim away elements not in the specified range. All other data remain unchanged, except thier indices will be adjusted such that the element in the 'start' postion will be translated to 0,0..0.

STensorVec slice(IntVec&);

Return only the specified elements.

STensorVec subTensorVec(const IntVec& start, const IntVec& len);
Return TensorVec, with it's own indexer, with the given dimensions.

Protected members
void verify(void) const;
Validate input.

Private members
TensorIndex idx;
Indexing function.

OPERATORS, FREE FUNCTIONS, CODE, ETC.
double dot(const STensorVec&, const STensorVec&); Dot product

INCLUDED FILES
iostream.h
Lattice/TensorIndex.h
SIndexedVec.h

NAME

SHypSurf – Parametric hyper-surface container class

DESCRIPTION

A parametric volume is a map from the plane to n space. Additions consist of curvature computing routines - Gaussian as well as maximal and minimal curvatures can be computed (including the associated directions).

PREREQUISITES

STools, VecMtx, RWTools

SEE ALSO

TPSurf

AUTHOR

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

CLASS SHypSurf

Base class(es)
public SError {

protected:
SHypSurf();

public:
virtual ˜SHypSurf();

virtual DoubleVec realEvaluate(DoubleVec& pt, IntVec& der) const =0;
virtual inline DoubleVec evaluate(DoubleVec& pt) const;
virtual inline DoubleVec evaluate(DoubleVec& pt, IntVec& der)const;

Evaluate the point on the surface at pt or evaluate the der-th order parametric derivative.

virtual void evaluate(DoubleVec* pts, IntVec& der, STensorVec* vals) const;
virtual void evaluate(DoubleVec* pts, STensorVec* vals) const;

Evaluate the surface along the hyper-grid formed by the elements of the pts vectors. Fill the vals structure with the details.

double safeStep(double x0, double x1, int nPts) const;

Determine the step required to get from x0 to x1 in n steps.

void gridEvaluate(DoubleVec& start, DoubleVec& stop, IntVec& steps,
STensorVec* vals);

Evaluate the surface along the hyper-grid formed by the elements of the vectors, v[i]. Each v[i] is computed as steps[i] equally spaced points on the interval (start[i]-stop[i]). The vals structure is filled with the evaluation results.

};

INCLUDED FILES

strstream.h
iostream.h
rw/dvec.h
rw/ivec.h

SIndexedVec/STensorVec.h
STools/SError.h

NAME

SHypBSurf – A container class for basis spline tensor product

DESCRIPTION

Bezier surfaces are derived from this.

NOTE

This is set up to evaluate the surface at the "last knot" as if that knot was the part of the previous segment (and not give back zero).

PREREQUISITES

Refiner, FuncBasis, ForwDiff, Tuple, NumberSequence, VecMtx, HTuple

SEE ALSO

SHypNUBSurf

AUTHOR

Bruce Hickey bhickey@watcgl.uwaterloo.ca

CLASS SHypBSurf

Base class(es)
public SHypSurf {

private:

protected:

unsigned numB;
The number of pointers in the following array.

BBasis **bBasisPntr;
Array of pointers to the bases.

unsigned spaceDegree;
This is the dimension of the space we expect.

void verify(void) const;
Verify the size of the control mesh and the basis dimensions correspond.

STensorVec cvMatrix;
The control point lattice.

inline void hook(unsigned, BBasis* nB);
An array of numB pointers to the bases. This routine must be called to hook this class' pointers to the bases in the derived class. These bases are not yet created at SHypBSurf construction time and consequently must be attached through this function. It must be called once for each basis.

public:
SHypBSurf();
SHypBSurf(int dim, IntVec& cmSizes, unsigned sd=3);
SHypBSurf(int dim, const STensorVec& m);
SHypBSurf(int dim, STensorVec& m);
These constructors are called by a derived class' constructor which has not recieved bases pointers in its argument list. These must be called in conjunction with init(). The spaceDegree of the last

two ctors is implied by the size of the STensorVec.

virtual ˜SHypBSurf();
Desctructor.

inline unsigned numBases() const;
Get the number of bases in *this.

virtual DoubleVec realEvaluate(DoubleVec& pt, IntVec& du) const;
inline DoubleVec realEvaluate(DoubleVec& pt) const;
Evaluate a point on the surface or a parametric derivative. This is called by evaluate routines. The second routine simply builds an empty IntVec and sends it to the first.

const BBasis& refBasis(unsigned n) const;
inline const BBasis refBasis(void) const;**
return a reference to the Nth basis, or all the bases at once.

virtual NumberSequence knots(unsigned n);
virtual NumberSequence breakpoints(unsigned n);
return the knot sequence for the Nth basis.

inline STensorVec& cvMat();
inline const STensorVec& cvMat() const;
Return the appropriately typed copy of the control mesh.

void cvMat(const STensorVec& x);
Set the entire control mesh. The sizes of x must agree with the bases.

inline DoubleVec cv(int);
inline void cv(int, const DoubleVec& newVert);
inline DoubleVec cv(IntVec&);
inline void cv(IntVec&, const DoubleVec& newVert);
Access or change a control vertex. The length of the DoubleVec is always spaceDegree

int dimension(unsigned n) const;
Get the size, in the Nth dimension, of the control vertex array. This is also the dimension of the Nth basis function.

IntVec dimensions(void) const;
Get the size control vertex array in each dimension. Again, this is the dimension of the basis functions.

int order(unsigned n) const;
IntVec order() const;
The order of the Nth basis/all bases.

virtual void printOn(ostream&) const;
virtual void scanFrom(istream&) ;
I/O routines.

};

ostream& operator<<(ostream&, const SHypBSurf&);

istream& operator>>(istream&, SHypBSurf&);

CLASS DECLARATIONS

class NumberSequence

INCLUDED FILES

iostream.h
rw/dvec.h
rw/dgemat.h
SIndexedVec/STensorVec.h
FuncBasis/BBasis.h
SHypSurf.h

NAME

SHypBezSurf – A Bezier hyper-volume

DESCRIPTION

Represents a Bezier tensor-product surface.

PREREQUISITES

Tuple, BezBasis, Refiner

SEE ALSO

SHypSurf, SHypBSurf

AUTHOR

T

CLASS SHypBezSurf

Base class(es)
public SHypBSurf {

public:

SHypBezSurf();
Default constructor.

SHypBezSurf(const IntVec& ords, const IntVec& numCPs, unsigned sd=3);
Specify the order and number of control points (numCPs[i]+1) in each dimension

SHypBezSurf(IntVec& ords, IntVec& dim, NumberSequence* bps, unsigned sd=3);
Specify the order/dimension/knots in each dimension, control points will be set to 0

SHypBezSurf(IntVec& ords, NumberSequence* bps, STensorVec& cvs);
Specify the orders/breakpoints in each dimension, control points are given explicitly.

SHypBezSurf(SHypBezSurf& x);
Copy constructor

virtual ˜SHypBezSurf();
Destructor.

virtual NumberSequence breakpointVec(int n);
Gets an alias to the knots of the Nth basis, so you can examine breakpoints and stuff. See NumberSequence for details about getting data out of the sequence. The simplest operation on a NumberSequence is accessing knots via knots(n)[i]. Often this is all you'll need.

SHypBezSurf& operator=(SHypBezSurf&);
Assignment operator, doesn't copy data. NOT IMPLEMENTED

BezSurf<NTuple, TupleMatrix> tpSlice(IntVec&);
Return a the BezSurf which lies in the indicated slice.

void rootSmooth(void);
Add breakpoints, based on "roots" to create a hyper-surface where each patch is "smoother"

void simpleSmooth(int numNewBps);
Add the given number of new breakpoints to each segment, in each dimension. This, in an effort to

make the patches smoother.

void addBreakpoint(double newBp, int dim);
void addBreakpoints(NumberSequence& newBps, int dim);
Refine the surface by adding the given breakpoint(s) in the given dimension.

virtual RWString className(void)const ;

virtual void printOn(ostream&) const;
virtual void scanFrom(istream&) ;
I/O routines.

protected:

const static RWString classname;
Class identifier.

BezBasis *bezBasisPntr;
Array of bases.

DoubleVec findRoots(int dim);
Find all the first derivative roots of the polynomial components of the spline hyper-volume, in the given dimension.

private:

};

ostream& operator<<(ostream&, const SHypBezSurf&);
istream& operator>>(istream&, SHypBezSurf&);

CLASSDOC ON

INCLUDED FILES
rw/ivec.h
FuncBasis/BezBasis.h
TPSurf/BezSurf.h
NumberSequence/NumberSequence.h
SHypNUBSurf.h
SHypBezSurf.in

NAME

SHypNUBSurf – Non-uniform B-spline Hyper volume

DESCRIPTION

Represents a non-uniform tensor product B-spline surface. A simple example of using SHypNUBSurf is in SHypNUBSurf.tst.cc. This program evaluates the surface and normals at a mesh of points, then renders using Renderer.

PREREQUISITES

Tuple, NUBBasis, Refiner, HTuple

SEE ALSO

SHypSurf, SHypBSurf

AUTHOR

T

CLASS SHypNUBSurf

Base class(es)
public SHypBSurf {

private:
protected:

NUBBasis *nubBasisPntr;
Array of bases.

public:

SHypNUBSurf();
Default constructor.

SHypNUBSurf(IntVec& ords, IntVec& numCPs, unsigned sd=3);
Specify the order and number of control points (numCPs[i]+1) in each dimension

SHypNUBSurf(IntVec& ords, IntVec& dim, NumberSequence* kts, unsigned sd=3);
Specify the order/dimension/knots in each dimension, control points will be set to 0

virtual ˜SHypNUBSurf();
Destructor.

//virtual inline NumberSequence knots(int n); Gets an alias to the knots of the Nth basis, so you can examine breakpoints and stuff. See NumberSequence for details about getting data out of the sequence. The simplest operation on a NumberSequence is accessing knots via knots(n)[i]. Often this is all you'll need.

virtual inline RWString className(void)const ;

virtual void printOn(ostream&) const;
virtual void scanFrom(istream&) ;
I/O routines.

NUBSurf<NTuple, TupleMatrix> tpSlice(IntVec&);
Return a the NUBSurf which lies in the indicated slice.
};

ostream& operator<<(ostream&, const SHypNUBSurf&);
istream& operator>>(istream&, SHypNUBSurf&);

INCLUDED FILES

rw/ivec.h
FuncBasis/NUBBasis.h
TPSurf/NUBSurf.h
SHypBSurf.h

NAME

BezCV2Poly – Generates a polynomial description of a Bezier.

DESCRIPTION

This class simply maintains a monomial conversion matrix. It allows the user to obtain the polynomials for a Bezier curve/surface segment.

```
BezCV2Poly conv(3); // The quartic Bezier
DoubleVec cvs(3,0,2); // {0, 2, 4}
DoubleVec coeffs= conv.getCoeffs(cvs); // The polynomial description
```

PREREQUISITES

Func, STools, rwtools, rwmath

SEE ALSO

Poly, BezCurve, BezSurf, SHypBezSurf

AUTHOR(S)

Bruce Hickey, bhickey@watpix

COPYRIGHT

CLASS BezCV2Poly

Base class(es)
virtual
public SError {

public:

BezCV2Poly(unsigned order);

Creates an instance which will convert Bezier control meshes of order "ord" to their polynomial representation.

DoubleVec getCoeffs(const DoubleVec& cvs, unsigned d=0);
TupleVec getCoeffs(const TupleVec& cvs, unsigned d=0);
DoubleGenMat getCoeffs(const DoubleGenMat& cvs, unsigned d=0);

Return the co-efficients of the polynomial with the given control net forms.

Poly getPoly(const DoubleVec& cvs);

Returns the polynomial definition.

unsigned order(void) const;

The order of the Bezier curve.

DoubleVec findAllRoots(TupleVec& cvs);

Finds all potential "problem" spots by computing the roots of the hyper-volumes first derivatives. This is done by computing polynomial components of the spline volume. The numRoots vector

will have the number of roots in each dimension as its contents. The returned numBases X maxNumRoots matrix will have the roots for each parameter domain as its entries.

virtual RWString className(void) const; Class id.

protected:
DoubleGenMat coeffMat;
Mononmial basis conversion co-efficients.

private:

};

inline members for class Stt

inline unsigned BezCV2Poly::order(void) const { return coeffMat.rows(); }
//inline RWString BezCV2Poly::className(void) { return "BezCV2Poly"; }

INCLUDED FILES
rw/dgenmat.h
rw/dvec.h
Tuple/TupleVec.h
Func/Poly.h
STools/SMath.h
STools/SError.h

NAME

SHypSurfFit – Container class for tensor product hyper-surface fitters.

DESCRIPTION

This is the equivalent of CurveFit class for the tensor product hyper-surfaces. Procedures that estimate the initial positions parameter values are available, as well as those to compute the residual and change the parameter values. Data points are in this class. This can only deal with "grided" parameter values (ie. the parameter values are two sequences, not a matrix). SHypSurfFitFree is included to have use of a "free standing" routines that are defined in there.

PREREQUISITES

SHypSurf,CurveFit,Refiner,FuncBasis,ForwDiff,NumberSequence,
Tuple,dspmat,VecMtx,linpack

SEE ALSO

SHypBSurfFit, CurveFit

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

BUGS

Routines changing parameter values may "change the order" of
the parameters. User has to decide what to do with these.

CLASS SHypSurfFit

Base class(es)
virtual
public SError {

private:
STensorVec dataToFit; Data matrix.

DoubleVec *parData;
This is an array of parameter point vectors. One for each degree. Holds the parameter values where the surface is to be evaluated to fit the data. Access via par() routine.

void getParSpace(unsigned);
Safely allocate space for the parameter vectors.

protected:
SHypSurf* defSurf;
Default surface. Access and set using defaultSurface() routines.

RWBoolean surfSetFlag;
Flag is true of pointer above is set.

RWBoolean mySurfCopy;
Flag is true if ownSurf() was used to indicate own copy of the surface.

virtual RWString className() const;
Returns above defined RWString.

void noSurfError(void) const;
Additional checks.

void removeSurf(void);
As we just keep pointer to the data and surface, we don't want the destructor to deallocate the memory. These functions do just that.

public:
virtual ˜SHypSurfFit();
Destructor.

SHypSurfFit();
SHypSurfFit(const SHypSurfFit&);
A default and copy constructor.

SHypSurfFit(IntVec& dims, unsigned tL);
Data matrix is tL x dims[0] x dims[1] x ...

SHypSurfFit(STensorVec& tv, DoubleVec *pars=NULL);
Construct with the data matrix with optional initial parameter values. The number of parameter DoubleVecs is implied by the dimensionality fo the STensorVec.

unsigned dimension(void) const;
The dimension of the data and the surface that will fit this data.

void changeData(const STensorVec&);
void changeData(const STensorVec&, DoubleVec*);
Change the data points and parameter values.

void ownSurf(void);
Indicate private copies of the data and default surface. Appropriate flags are set; there is no actual data copying going on.

const STensorVec& data(void) const;
STensorVec& data(void);
Data points.

unsigned length(unsigned);
const IntVec& lengths(void) const;
int tupleLength(void) const;
Information about the data matrix.

const DoubleVec* par(void) const;
DoubleVec* par(void);
const DoubleVec& par(unsigned) const;
DoubleVec& par(unsigned);
Access parameter values used in the fitting process.

void par(unsigned, DoubleVec&);
Access parameter values in a given dimension

SHypSurf* defaultSurface(void);
const SHypSurf* defaultSurface(void) const;
Look at the default surface.

SHypSurf& refDefSurface(void);

const SHypSurf& refDefSurface(void) const;

int defaultSurfaceSet(SHypSurf*);
int defaultSurfaceSet(void) const;
First one sets the default surface and returns the previous value of the surfaceSetFlag; the second one returns the value of the surfaceSetFlag.

double residual(void);
double residual(const SHypSurf& srf);
Total residual divided by the number of points.

void residual(STensorVec*, RWBoolean flag=FALSE);
void residual(const SHypSurf& srf, STensorVec*, RWBoolean flag=FALSE);
Matrix of residual vectors (if the flag is false) or residual values (if the flag is true) at each of the data points.

void residual(DoubleVec*);
void residual(const SHypSurf& srf, DoubleVec*);
Total residual at each "row" and each "column" of the data points.

void printOn(ostream&) const;
void scanFrom(istream& is);
};

ostream& operator<<(ostream& strm, const SHypSurfFit& x);
istream& operator>>(istream& is, SHypSurfFit& x);

INCLUDED FILES
rw/ivec.h
SHypSurf/SHypSurf.h
SIndexedVec/STensorVec.h
STools/SError.h
SHypSurfFitFree.h

NAME

SHypBSurfFit – Fit data with B-spline tensor product surface.

DESCRIPTION

This is the equivalent of BCurveFit class for the tensor product surfaces. The goal is to compute the least square fitted B-spline tensor product surface to a matrix of data points. The fit is a two step process, and two steps can be applied in any order. There is a procedure that, given the basis and parameter values corresponding to the data points will compute the control points of the surface that fits those points in the least squares sense. Another procedure will take a surface and change the control points to reduce the residual difference. Procedures that estimates the initial positions parameter values are available.

PREREQUISITES

TPSurf,CurveFit,Refiner,FuncBasis,ForwDiff,NumberSequence,
Tuple,dspmat,VecMtx,linpack

SEE ALSO

BSurf, TPSurfFit, CurveFit

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

BUGS

There is no "leave-interpolate" version yet. There must be an equal number
of data and parameter points in each dimension.

CLASS SHypBSurfFit

Base class(es)
public SHypSurfFit {

private:
protected:
virtual RWString className() const;
Used for error messages.

public:
void printOn(ostream&) const;
void scanFrom(istream&);

˜SHypBSurfFit();
Destructor.

SHypBSurfFit();
SHypBSurfFit(const SHypBSurfFit&);
A default and copy constructor.

SHypBSurfFit(IntVec& dims, unsigned tL);
Data matrix is tL x dims[0] x dims[1] x ...

SHypBSurfFit(STensorVec& tv, DoubleVec* pars=0);
Construct with the data matrix and initial parameter values.

SHypBSurf* defaultSurface(void);
const SHypBSurf* defaultSurface(void) const;
Return the pointer to the default surface.

SHypBSurf& refDefSurface(void);
const SHypBSurf& refDefSurface(void) const;

Return the reference to the default surface.

virtual DoubleVec startKnots(NumberSequence* ns,
IntVec& numIntervals,
IntVec& orders);

This will compute the initial knot sequence if you have no better way of figuring that out. The decision will be based on the number of data points. NumberSequences passed should be empty, and the dimension()*2 element DoubleVec that is returned are the values that should be used as bounds for the parameter values. These are alternate pairs of start/stop values for parameter sequences. numIntervals is the required number of intervals where the surface is to be "fully" defined.

virtual void startKnots(NumberSequence* ns, DoubleVec*,
IntVec& numIntervals,
IntVec& orders);

As above, but in this case the interval where the surface is "fully" defined is passed as the argument - not returned.

int lsFit(void);
int lsFit(SHypBSurf*);
STensorVec lsFit(const BBasis bases);**
int lsFit(const BBasis bases, STensorVec*);**

Compute the control points to make the resulting tensor product surface (on given basis) best (least squares) fit to the data points. First deals with the default surface, first and second will change the control vertices of the surface (as oppose of just returning them, as in the last two).

void findTopAndSizes(IntVec&, IntVec&, IntVec&, IntVec&);
void findSandECols(const BBasis&, IntVec&, IntVec&, const DoubleVec&,
int);

int intrpLsFit(const IntVec&, const IntVec&);
int intrpLsFit(BSurf*, const IntVec&, const IntVec&);
TupleMatrix intrpLsFit(const BBasis&, const BBasis&,
const IntVec&, const IntVec&);
int intrpLsFit(const BBasis&, const BBasis&, TupleMatrix*,
const IntVec&, const IntVec&);

Do a least squares after attepmting to interpolate points defined by a cross product of the two IntVecs.

int leaveLsFit(const IntVec&, const IntVec&);
int leaveLsFit(BSurf*, const IntVec&, const IntVec&);

This will constraint the least squares process to change only the rows of control vertices corresponding to the zero elements of the first IntVec and columns corresponding to the zero elements of the second IntVec. NOTE : This is not appropriate as the "first" computation of the control vertices.

int leaveIntrpLsFit(const IntVec&, const IntVec&,
const IntVec&, const IntVec&);
int leaveIntrpLsFit(BSurf*,
const IntVec&, const IntVec&,
const IntVec&, const IntVec&);

First two IntVecs are used for the "leave" part of the least squares process, the second two for the "interpolate" part. WARNING : Not implemented yet.

IntVec bestSimpleInsert(double*, double*, double tol=0.0);
IntVec bestWeightedInsert(double*, double*, double tol=0.0);
IntVec bestSimpleInsert(const BSurf&, double*, double*, double tol=0.0);
IntVec bestWeightedInsert(const BSurf&, double*, double*, double tol=0.0);

Using the simple minded algorithms, this will provide the knots that should be inserted in u and v direction to improve the fit. It will fail if inserting the knot at the required place will force discontinuity. IntVec returned has two elements, first one being the success flag for U, the second one for V direction.

};

ostream& operator<<(ostream& strm, const SHypBSurfFit& x);
istream& operator>>(istream& is, SHypBSurfFit& x);

INCLUDED FILES

SHypSurf/SHypBSurf.h
SHypSurfFit.h

NAME

SHypNUBSurfFit – Fit data with non-uniform B-spline hyper-surface.

DESCRIPTION

PREREQUISITES

SHypSurf, CurveFit, Refiner, FuncBasis, ForwDiff, NumberSequence, Tuple, rwtool, rwmatx, rwmath

SEE ALSO

NUBSurf, BSurfFit, TPSurfFit, CurveFit

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

CLASS SHypNUBSurfFit

Base class(es)
public SHypBSurfFit {

private:

protected:
virtual RWString className() const;
Used for error messages.

public:
˜SHypNUBSurfFit();
Destructor.

SHypNUBSurfFit();
SHypNUBSurfFit(const SHypNUBSurfFit&);
A default and copy constructor.

SHypNUBSurfFit(IntVec& dims, unsigned tL);
Data matrix is tL x dims[0] x dims[1] x ...

SHypNUBSurfFit(const STensorVec& tv);
Construct with the data matrix.

SHypNUBSurfFit(STensorVec& tv, DoubleVec* pars=NULL);
Construct with the data matrix and initial parameter values.

SHypNUBSurf* defaultSurface(void);
const SHypNUBSurf* defaultSurface(void) const;
Return the pointer to the default surface.

SHypNUBSurf& refDefSurface(void);
const SHypNUBSurf& refDefSurface(void) const;
Return the reference to the default surface.

void printOn(ostream&) const;
void scanFrom(istream&);
};

ostream& operator<<(ostream& strm, const SHypNUBSurfFit& x);
istream& operator>>(istream& is, SHypNUBSurfFit& x);

INCLUDED FILES

SHypSurf/SHypNUBSurf.h
SHypBSurfFit.h

NAME

SHypSurfFitFree – "Free" functions to help surface fitting.

DESCRIPTION

PREREQUISITES

CurveFit,Tuple,Linalg,Mat,VecMtx,linpack

SEE ALSO

CurveFitFree, TPSurfFit, CurveFit

AUTHOR

Bruce Hickey, bhickey@watpix.uwaterloo.ca

OPERATORS, FREE FUNCTIONS, CODE, ETC.

int unconstrLs(SCompMat* evalMat, unsigned num, STensorVec& P, STensorVec* V);

Compute the least squares solution to the system

$$\min \| B\, V^t\, C - P \|$$

using Kronecker products to simplify the procedure.

int constrLs(const DoubleGenMat& B, const DoubleGenMat& C,
const TupleMatrix& P,
const DoubleGenMat& K, const DoubleGenMat& L,
const TupleMatrix& D,
TupleMatrix* V);

Compute the least squares solution to the system

$$\min \| B\, V^t\, C - P \| \text{ subject to } K\, V^t\, L = D$$

using Kronecker products to simplify the procedure.

void fitter (STensorVec*, STensorVec*, SLeastSqHH*, int dim);

INCLUDED FILES

rw/dgemat.h
CurveFit/CurveFitFree.h
SLinAlg/SLeastSqHH.h
SIndexedVec/SCompMat.h
SIndexedVec/STensorVec.h

NAME

SLeastSqHH – Least squares solver for SCompMat objects

DESCRIPTION

An object of this type will solve the least squares problem

```
A * solution ~ b
```

It is assumed that the columns of [A] are linearly independent. This object performs the same function as a rouge wave DoubleLeastSq object. The information contained in the SCompMat about the existence of zeros in the matrix A is used to minimize the factorization process. If linear dependence is detected, based on the tolerance setting, the value of isUnderDetermined will be TRUE. Subsequent calls to solve will fail and return FALSE.

PREREQUISITES

SIndexedVec, Lattice, Tuple, STools, rwtool, rwlpak, rwmatx, rwmath

SEE ALSO

CurveFit, TPSurfFit, SHypSurfFit

AUTHOR(S)

Bruce Hickey, bhickey@watcgl

COPYRIGHT

CLASS SLeastSqHH

Base class(es)
public SError

Public members
SLeastSqHH();
SLeastSqHH(SLeastSqHH&);
SLeastSqHH(const SCompMat&);

virtual ~SLeastSqHH();

void factor(const SCompMat&);

The given matrix factored into the form QR using Householder transformations: H = I + SCAL*V*(V**T). The i-th vector V for Q is stored from the 0-th position to the i-th position in the rows-i-th row of matrix, and the i-th scale factor, SCAL, his stored in the array named scal. The vectors V for P are stored in the left-hand column positions of the A portion of the array matrix. The scale factors for P are also stored in the array named scal. The matrix passed to this function will be copied then factored. Thus the parameter matrix is unaltered. This is opposite to the constructor which uses an alias and factors the parameter matrix.

DoubleVec solve(const DoubleVec& rhside);
void solve(const TupleVec& rhside, TupleVec* solution);

TupleVec solve(const TupleVec& rhside);
Backsolve the solution to a factored system with a single or multiple right-hand sides.

DoubleVec residuals(DoubleVec& rhside);
void residuals(TupleVec& rhside, TupleVec* resids);
Perform the residuals calculation.

RWBoolean SLeastSqHH::isUnderDetermined(void);
Tests if the factored matix is under-determined with respect to the tolerance value.

double singularTest(void);
Uses the current factored matrix to find a solution, x, to Ax = R, where R is a vector of random (0-1) numbers. The largest value of the solution is returned. This number should be realtively small if the matrix is non-singular.

void SLeastSqHH::setTolerance(double);
Set the tolerance within which a factorization will find a matrix under-determined. Initially set to the literal SEPS, found in <numerics.h>

const DoubleVec& diagonal(void) const;
const DoubleVec& scale(void) const;
Obtain information on the factorization, useful when factorization is unsuccessful.

virtual void printOn(ostream&) ;
virtual void scanFrom(istream&);
I/O functions

Protected members
RWBoolean isFactored;
RWBoolean isUnderDet;
DoubleVec diag;
DoubleVec scal;
double toler; The current tolerance.

SCompMat matrix;

RWBoolean doFactor(void);
Perfrom actual factoring.

Private members

OPERATORS, FREE FUNCTIONS, CODE, ETC.
ostream& operator<<(ostream&, SLeastSqHH&);
istream& operator<<(istream&, SLeastSqHH&);

INCLUDED FILES
math.h
numerics.h
iostream.h
rw/dvec.h
Tuple/TupleVec.h
SIndexedVec/SCompMat.h

STools/SError.h

NAME

SHypSurfMinim – Provides miminimization routines for SHypSurfs.

DESCRIPTION

Minimize nonlinear sum of residual squares using analytic jacobian. You can invoke the local minimizer with your own initial point or allow the class to use it automatic initial point generation routines. The actual minimization routines were written by John E. Dennis, Jr., David M. Gay, and Roy E. Welsch from "Acm Transactions on Mathematical Software", September, 1981.

```
SHypNUBSurf aSurface;
DoubleVec aPoint;
DoubleVec initalGuess;
SHypSurfMinim minimizer(&aSurface);
DoubleVec domainSoln = minimizer.closest(initalGuess);
DoubleVec closestPoint = aSurface.evaluate(domainSoln);
```

PREREQUISITES

SHypSurfMinim, SHypSurf, TPSurf, ForwDiff, Refiner, FuncBasis, SIndexedVec, NumberSequence, SAffGeom, Tuple, STools, rwtool, rwmath, f2c

SEE ALSO

SHypBezSurf

AUTHOR(S)

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

COPYRIGHT

CLASS SHypSurfMinim

Base class(es)
public virtual SError

Friends

ostream& operator<<(
ostream& os,
const SHypSurfMinim& inst
);

No input operator is provided as this is an actor class on SHypSurfs and maintains no persistant state.

Public members

enum searchMeth { SAMPLE, USECVS };

The possible methods for the initial point computation.

SHypSurfMinim(void);

A default (no arguments) constructor is included so we can create arrays of SHypSurfMinim.

SHypSurfMinim(const SHypSurfMinim& rhs);
Copy constructor.

SHypSurfMinim(SHypBSurf* refSurf);
Provide a pointer to the surface which we will be working on.

˜SHypSurfMinim(void);
Destructor

void defSurf(SHypBSurf* refSurf);
Provide a pointer to the surface which we will be working on. No provision is made to protect any currently referenced surface.

DoubleVec closest(
DoubleVec* u, Initial domain space guess
DoubleVec& p External point
);
Determine the closest point to the surface, to p, given an initial guess, u. The parameter values which yield this surface point are returned in u.

DoubleVec closest(
DoubleVec& p, External point
searchMeth sm=SAMPLE ,
DoubleVec* domPt=0 domain point providing best soln
);
Determine the closest point on the surface.

virtual RWString className(void) const;
Get an identifier.

unsigned dataDim(void) const;
The dimension of the data.

unsigned domainDim(void) const;
The domain of the spline.

void evaluate(DoubleVec& u);
Evaluate the underlying surface at the given point.

virtual void printOn(ostream& os) const;
virtual void scanFrom(istream& is);
I/O routines.

RWBoolean inValidInterval(const DoubleVec& x);
TRUE if x is in the valid interval of the underlying surface.

HERE FOR DEBUGGING PURPOSES ONLY !!!!
SHypBSurf* theSurf; The underlying surface

Protected members

const static RWString classname;
Most classes should have a unique identifier.

void verify(void) const;

DoubleVec closestBruteForce(DoubleVec& p, DoubleVec* u);
This procedure will perform the minimization with many inital guesses and return the best result. Currently this scheme will only work if the underlying surface's control net is convex.

DoubleVec closestUsingCVs(DoubleVec& p, DoubleVec* u);
This procedure will perform the minimization with an inital guesses midway, in the parameter domain, between the closest CV to the external point and the diagonally opposite CV. As an example, the CV (4,5,6,8) would have (3,4,5,7)=(4-1,5-1,6-1,8-1) as its diagonal opposite. Currently this scheme will only work if the underlying surface is a SHypBezSurf.

Private members

DoubleVec surfPt; The current surface point
DoubleGenMat derVals; The first derivatives at the above point

DoubleVec vv; Work arrays needed by nl2sol
IntVec iv, ui; Work arrays needed by nl2sol

Globals used for data transfer to C routines

EXTERNAL DECLARATIONS

double* srf_glob;
double* ders_glob;
SHypSurfMinim* theMinimizer;

INCLUDED FILES

iostream.h
rw/dvec.h
rw/dgemat.h
SHypSurf/SHypBSurf.h
STools/SError.h

Bibliography

[1] Apple Computer Corp. *LaserWriter User's Guide*, 1991.

[2] James Avro, editor. *Graphics Gems II.* Academic Press, 1991.

[3] R.E Barnhill and Riesenfeld R.F. *Computer Aided Geometric Design.* Addison Wesley, 1974.

[4] R. Bartels, J. Beatty, and B. Barsky. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[5] I.E. Bell. *Gamut-Mapping Algorithms for Reflective Image Representations.* PhD thesis, University of Waterloo, Waterloo, Ontario, In progress.

[6] P. Bézier . Définition numérique des courbes et surfaces i. *Automatisme*, 11:625–632, 1966.

[7] R.P. Brent. *Algorithms for Minimization without Derivatives.* Prentice-Hall, 1973.

[8] S.D. Conte and C. de Boor. *Elementary Numerical Analysis.* McGraw-Hill, 1980.

[9] C. de Boor. A practical guide to splines. In *Applied Mathematical Sciences*. Springer-Verlag, New York, New York, 1978. Volume 27.

[10] John E. Jr. Dennis, David M. Gay, and Welsch Roy E. An adaptive nonliner least-squares algorithm. *ACM Transactions on Mathematical Software*, 7(3):348–368, 1981.

[11] P. Dierckx. An algorithm for least-squares fitting of cubic spline surfaces to functions on a rectilinear mesh over a rectangle. *Journal of Computational and Applied Mathematics*, 2(3):113–129, 1977.

[12] P Dierckx. *Curve and Surface Fitting with Splines*. Clarendon Press, 1993.

[13] M.A. Ellis and B. Stroustup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[14] G.E. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, Inc., 1990. Second edition.

[15] James D. Foley, Andries. van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison Wesley, 1990.

[16] David R. Forsey and Richard H. Bartels. Tensor products and hierarchical fitting. *SPIE, Curves and Surfaces in Computer Vision and Graphics*, 1610:88–96, 1991.

[17] Andrew Glassner, editor. *Graphics Gems*. Academic Press, 1990.

[18] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1989. Second edition.

[19] M. Grossman. Parametric curve fitting. *Computer Journal*, 14(2):169–172, 1970.

[20] H. Inoue. A least-squares smooth fitting for irregularly spaced data: Finite-element approach using the cubic b-spline basis. *Geophysics*, 51(11):1051–2066, 1986.

[21] G.H Joblove and D.P Greenberg. Colour spaces for computer graphics. *Computer Graphics*, 3:20–25, 1978.

[22] F.L. Kitson. An algorithm for curve and surface fitting using b-splines. In *International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages 1207–1210, 1989.

[23] P. Lancaster and K. Šalkauskas. *Curve and surface fitting: An introduction.* Academic Press, Inc., 1986.

[24] D.G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1973.

[25] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[26] National Institutes of Health. *NIH Class Reference*, 1989.

[27] W.A Paeth. Fast algorithms for color correction. *Proceedings of the Society for Information Display*, 30(3):169–175, 1989.

[28] W.A. Paeth. *Digital Models for Subtractive Colour.* PhD thesis, University of Waterloo, Waterloo, Ontario, In progress.

[29] Rogue Wave Software. *Math.h++ Introduction and Reference Manual*, 1991.

[30] Rogue Wave Software. *Matrix.h++ Introduction and Reference Manual*, 1991.

[31] Michael W. Schwarz, William B. Cowan, and John C. Beatty. An experimental comparison of RGB, YIQ, LAB, HSV, and opponent color models. *ACM Transactions on Graphics*, 6(2):123–158, 1987.

[32] M. Srećković. Adaptive hierarchical fitting curves and surfaces. Master's thesis, University of Waterloo, 1992.

[33] Maureen C. Stone, William B. Cowan, and John C. Beatty. Color gamut mapping and the printing of digital color images. *ACM Transactions on Graphics*, 7(4):249–292, 1988.

[34] B. Stroustup. *The C++ Programming Language, 2nd Edition.* Addison-Wesley, 1991.

[35] A. Vermeulen and R. Bartels. C++ splines classes for prototyping. *Curves and Surfaces in Computer Vision and Graphics II*, 1610:121–131, 1992. SPIE Proceedings, SPIE '92, Bellingham, Washington.

[36] Wavefront Technologies Ltd. *Data Visualizer User's Guide*, 1991.