Dyadic Splines

By
Mark Alan Duchaineau

Department of Computer Science
University of California, Davis
May, 1996

Dyadic Splines

By

Mark Alan Duchaineau

B.S. (California State University, Hayward) 1987
M.S. (University of California, Davis) 1990

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in Charge

May, 1996

Acknowledgements

This dissertation exists only because of the support of many people. The first tentative steps along these lines of research began when my friend David Thomas gave me a stack of periodicals containing early work in fractals. Inspired, Y. Rorke Weigandt and I worked in parallel and we both managed to synthesize terrain. My undergraduate mentor, Christopher Morgan, gave me the opportunity to develop a more logical set of subdivision rules and encouraged my various research pursuits. His enthusiasm, generousity and depth of mathematical insight gave me the skills, confidence and opportunities to continue my research at the graduate level.

Ken Joy took me into his fledgling computer graphics research group and has been untiring in his support throughout my years in Davis. The worlds of geometric modeling and image synthesis came alive as never before through his lectures and collaboration. The essences and culture of these fields came through clearly, and I have taken them to heart. At every step on the long road leading to this dissertation, Ken has been a staunch and skilled supporter.

The Davis graphics group has included some exceptionally talented students over the years who have helped me in numerous ways. Jim Reus has always been a fount of all knowledge, and a model of dedication to the myriad projects that whirl around him. Mark Miller was the first user of these ideas, helping me to clarify the concepts and make them work in the "real world." He did the first research into compression using dyadic splines [30]. Carlos Borges gave me an explanation of least-squares fitting that resonates in me to this day. All three waded through early manuscripts of this work and provided thoughtful suggestions.

Many other students and faculty members have read pieces of this work, attended my talks and examined my demonstrations. Their feedback has been invaluable. I would particularly like to thank Bob Estes, Ben Garlick, Drew Harrington, Peter Linz, Nelson Max, Richard Moster, Todd Reed and Stan Stoneking.

On a more personal note, my family has stood by me throughout every trial. Their unfailing love and boundless faith in me has kept my health sound and spirits high.

Lastly, I dedicate this dissertation to the late Paul Andersen, who guided me back to the high road.

Contents

Chapter 1

Introduction

Dyadic splines are a simple and efficient function representation that supports multiresolution design and analysis. These splines are defined as limits of a process that alternately doubles and perturbs a sequence of points, using B-spline subdivision to smoothly perform the doubling. An interval-query algorithm is presented that efficiently and flexibly evaluates a limit function for points and intervals. Methods are given for fitting these functions to input data, and for minimizing the energy and redundancy of the representation. Several methods are given for designing dyadic splines by controlling the perturbations of the limit process. Several applications are explored, including shape design, synthesis of terrain and other natural forms, and compression.

1.1 Overview

Function representations are the foundation of systems to model geometry and light interactions. As geometric scenes and lighting effects more closely resemble the complex world around us, there is an increasing need to reference their defining functions at multiple levels of detail, and to construct the functions only in neighborhoods that significantly impact the result. Furthermore, increasingly large datasets must be analyzed and stored compactly with a graceful loss of accuracy. Finally, designers of functions want intuitive controls that have desirable effects at appropriate scales and localities. Dyadic splines facilitate these goals.

The idea of doubling and perturbing is depicted in Figure 1.1.1. An initial sequence of points is doubled using weighted averages. Each point is split into two children. The left and right children are initially set to be one quarter of the way from the parent to the parent's left or right neighbor, respectively. This is the doubling step with weights that will be seen to produce smoothly-joined quadratic pieces in the limit function. The perturbation step is shown next. Each of the points produced by the doubling step is moved by adding some displacement to its position. It is these displacements that provide design handles and fitting parameters to the dyadic-spline function. This process is repeated, resulting in the limit function shown.

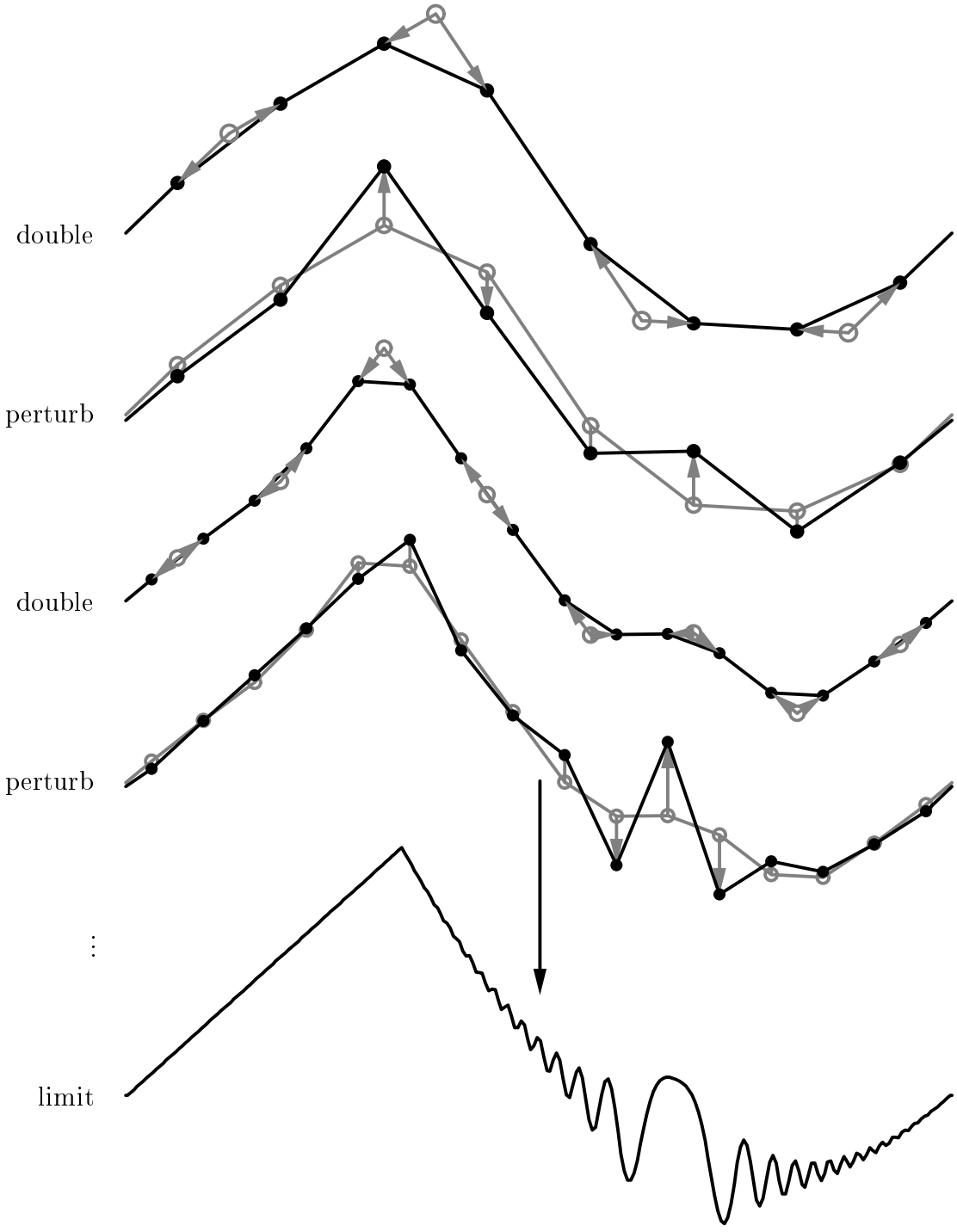


Figure 1.1.1: doubling and perturbing

This limit function exhibits the generality of dyadic splines. It has continuous derivatives of all orders except at a third of the way from the left, where there is a discontinuity in the first derivative. The first third of this function is linear, and the remainder is a transcendental function.

It is the perturbations that are the key to the generality exhibited in the example. The doubling process by itself produces quadratic pieces that are joined with first-order continuity (other weighting schemes can produce higher degree pieces that join more smoothly). By adding perturbations, limit functions may contain various types of discontinuities, may be nowhere differentiable (e.g. *fractals*), or may be transcendental functions. Indeed, all of the functions used in geometric modeling and image synthesis can be represented by dyadic splines.

The power of the doubling process comes from maintaining a sequence of approximations and in predicting finer approximations from coarser ones. The sequence of approximations double in resolution at each step. By maintaining copies of the function at each resolution, dyadic splines become an example of a *multiresolution* representation. Multiresolution representations allow geometry, texture and lighting computations to be carried out using optimum levels of detail.

The predictive power of the doubling process allows function energy to be pushed to coarse resolutions. This improves the accuracy of the approximations early in the doubling process, and facilitates compression of the function data. This prediction process is termed a *multiresolution analysis*. Figure 1.1.2 shows the multiresolution analysis for the limit function just given. On the left of the figure the sequence of perturbations are shown. It turns out that for special prediction processes that give perfect predictions whenever possible, the perturbations are redundant and contain twice as many values as needed. The right side of Figure 1.1.2 shows the perturbations after this redundancy has been removed.

In the dyadic-spline construction the simplest design handles are provided by the perturbations. Two curve modifications are depicted in Figure 1.1.3. The first creates a sharp indentation, while the second causes a broad rise. Clearly it would be tedious or impossible to achieve these effects with direct edits to a single-resolution representation. The term *multiresolution design* is introduced here for this kind of editing.

More sophisticated forms of multiresolution design are possible. For example, the shape of a specific tool could be used to sculpt the function, as shown in Figure 1.1.4. This is an example where the perturbations are controlled *indirectly* by the designer's actions.

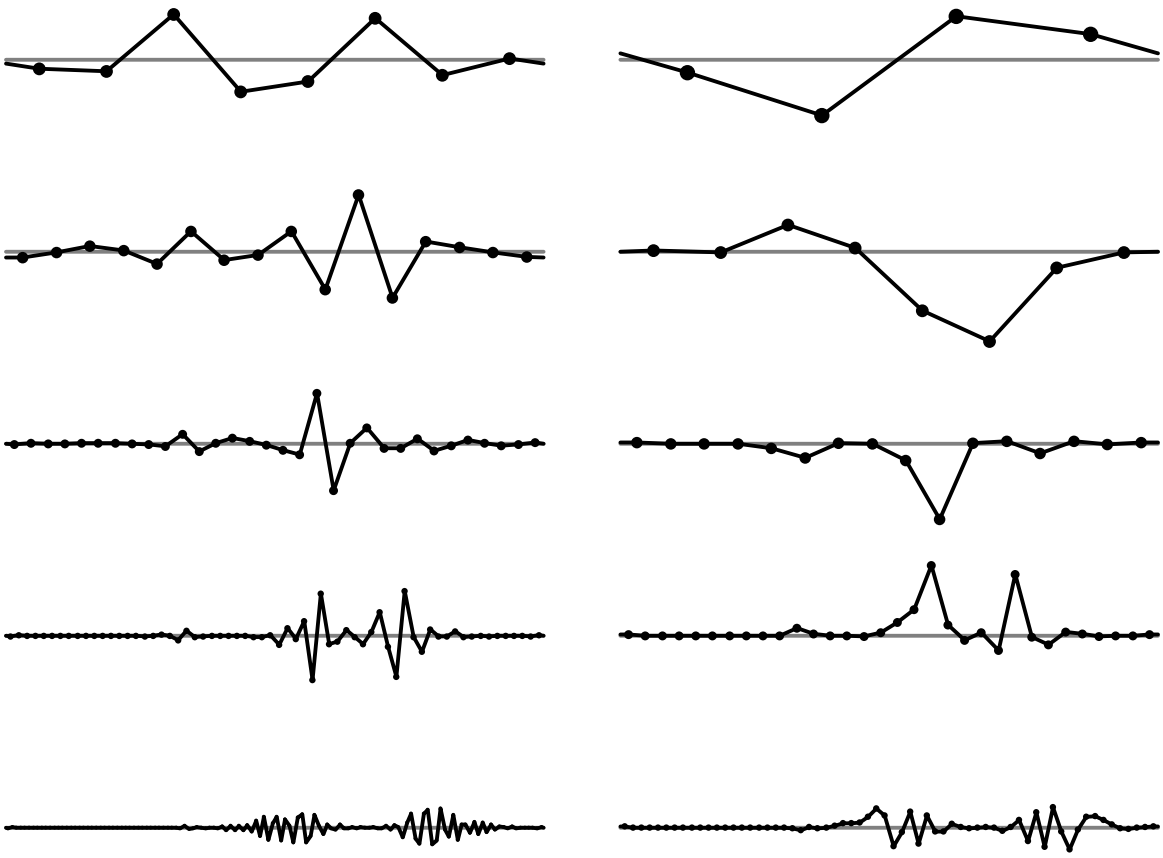


Figure 1.1.2: multiresolution analysis
(a) perturbations (b) redundancy removed

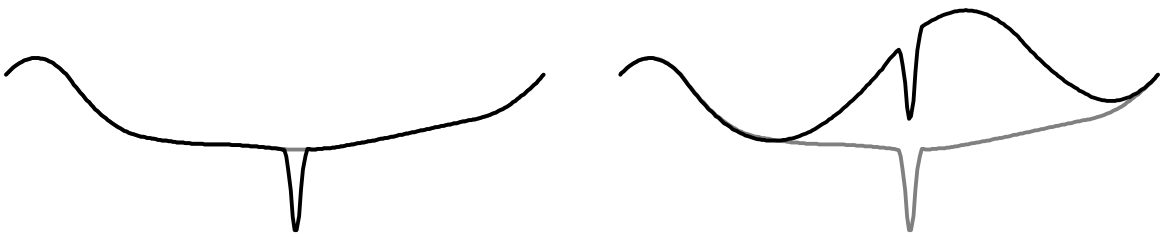


Figure 1.1.3: direct multiresolution edits

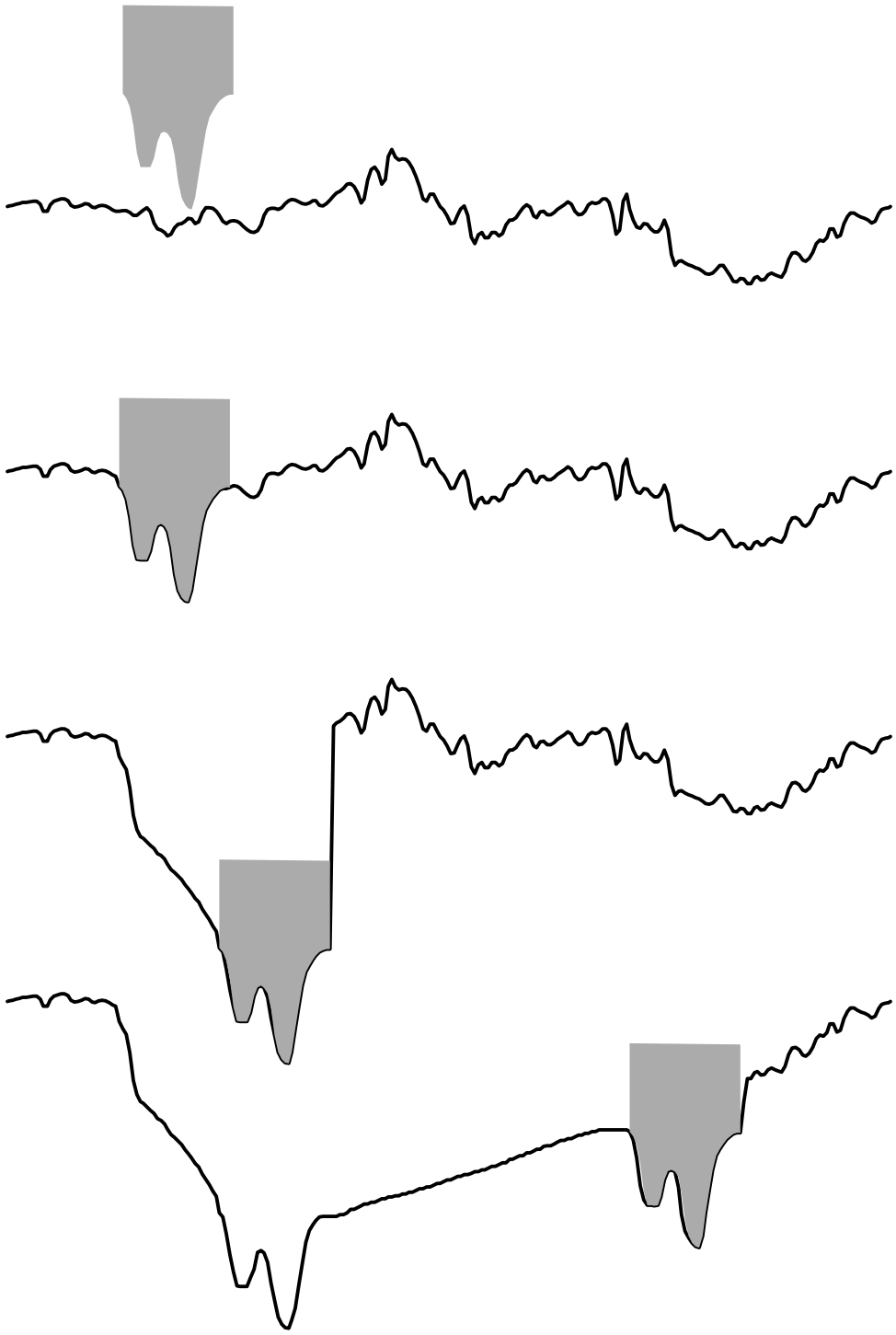


Figure 1.1.4: sculpting with a tool

1.2 Dissertation Outline

The remainder of this dissertation fleshes out the ideas outlined above. The next chapter gives detailed background on the representations most closely related to dyadic splines, namely bintrees, B-splines, hierarchical B-splines, wavelets and fractals.

Chapter 3 describes the *synthesis* of a limit function from the perturbations and doubling rules that define it. The double/perturb limit process is first defined mathematically, and then an *interval-query* evaluation method is introduced. This evaluation mechanism provides an efficient and flexible means for applications to access the function at multiple resolutions.

Prediction and analysis is covered in Chapter 4. The primary tools described in this chapter are: (1) a bottom-up process for converting high-resolution uniform B-splines into the multiresolution form of dyadic splines, (2) a process of *compacting* the dyadic-spline perturbations to remove a factor-of-two redundancy, and (3) a modification to the conversion and compaction operations to allow efficient computation.

Chapter 5 describes two algorithms for approximating target functions with dyadic splines. The first algorithm is a direct application of the simple bottom-up analysis process. Starting from detailed knowledge of the target function, the perturbations with smallest magnitudes are removed in order to decrease data complexity. This bottom-up approximation algorithm runs in time that is proportional to the amount of detailed data provided for the target function. The second algorithm uses advanced techniques to increase data complexity in a top-down fashion until the approximation is within tolerance. The running time is proportional to the complexity of the output approximation. This top-down algorithm can be hundreds of times faster than the bottom-up algorithm.

Curve editing techniques are described in Chapter 6 which make extensive use of the synthesis, analysis and approximation algorithms. The dyadic-spline representation provides a unifying methodology for describing a wide variety of possibilities for curve design. The design mechanisms discussed are: displacement editing, local smoothing, local roughening, offset-frame displacement editing, template edits and sculpting.

The extension to multiple variables is given in Chapter 7. This extension is based on the tensor-product B-splines. The univariate synthesis, analysis, approximation and design techniques have natural counterparts in the tensor-product setting. Also discussed is the application of dyadic-spline fractals to modeling natural phenomena. Finally, Chapter 8 concludes the dissertation. A summary of the work is given, including a discussion of the strengths and weaknesses of the dyadic-spline representation and an outline of future research directions.

Chapter 2

Related Representations

In this chapter, various representations are reviewed that have inspired the dyadic-spline formulation. These representations are outlined in some depth, and their desirable features are identified. In the case of bintrees, the basic principle of splitting up space into well-organized pieces is of essential value. B-splines provide a means of approximating a wide variety of functions while facilitating refinement, differentiation and bounding. Hierarchical B-splines provide a means of designing functions at different levels of detail simultaneously, and allow multiresolution fitting that brings all points to within tolerance. Wavelet multiresolution analysis provides a means of approximating a wide variety of functions in a manner that facilitates compression of control vectors and the solution of physical equations. Finally, fractals allow natural multiresolution detail to be added automatically, thus providing what has been termed *data amplification*. The dyadic splines will make use of the features of these representations.

2.1 Space Trees

In modeling and rendering, numerous schemes are used for decomposing space into disjoint pieces. These include quadtrees, octrees, bintrees, k-d trees and BSP trees [33]. Collectively, such methods are herein referred to as *space trees*. Space trees do not by themselves represent continuous-valued functions, but instead represent sets or partitions of space. The space-tree methods form excellent spatial organizations within which functions can be represented.

All space-tree representations have several properties in common. They consist of a tree where each node represents a subset of space. The children of a node are disjoint, and their union is the parent. Each leaf node is labeled with a partition index, which in the simplest case just indicates whether the leaf is in or out of a set. For each node, there is a simple means of deciding which child a point resides in. The tree is constructed in its entirety before being used, and the means of construction is "forgotten."

Space trees are useful for compression of uniform partitions since children with the same label can be merged into a single parent. These trees allow additional detail to be added just where it is needed. Space trees are also useful for speeding up geometric searches and for providing useful orderings of space.

By definition, a partition is a complete, disjoint collection of subsets of a space X. In mathematical notation, a partition is expressed as $\{X_i \subset \Re^n \mid i = 1, \ldots, m\}$, where $X = \bigcup_i X_i$ and $X_i \cap X_j = \emptyset$ for $i \neq j$. If each of the children X_i of X are recursively partitioned, then a general tree is formed. The mathematical notation for this is more readable if a child indexing function $c(i, j)$ is used to reference the jth child of node X_i. The function $j_{\max}(i)$ will indicate the number of children that X_i has. With these functions, the general tree partition is then denoted

$$X_i = \bigcup_{j=1}^{j_{\max}(i)} X_{c(i,j)}$$

where $X_{c(i,j)} \cap X_{c(i,k)} = \emptyset$ for $j \neq k$. The root node is $X_0 = X$.

The remaining mathematical notation regards the partition labeling functions. For each nonleaf node X_i, the function $p_i : X_i \to \{1, \ldots, j_{\max}(i)\}$ is given as

$$p_i(x) = c(i, j) \quad \text{for the unique } j \text{ such that } x \in X_{c(i,j)}$$

This determines which child $X_{c(i,j)}$ of X_i contains the point x. This can be applied recursively to obtain the leaf node $X_{p(x)}$ that contains x:

$$p(x) = q(0, x)$$

where

$$q(i, x) = \begin{cases} i & \text{if } X_i \text{ is a leaf} \\ q(p_i(x), x) & \text{otherwise} \end{cases}$$

Instances of this abstract method depend on how nodes are split up into children. The most general technique uses arbitrary linear halfspaces. The space trees formed this way are called Binary Space Partition (BSP) trees. At the other extreme are the quadtrees and octrees, where nodes are squares and cubes respectively with four and eight children having the same shape as the parent but with smaller dimensions. The k-d trees are similar to BSP trees, but with the normal to the halfspace always parallel to an axis (i.e. $x, y, \ldots$). Finally, bintrees are like k-d trees but where splitting is always done in the center, and where each axis is split in order, cycling repeatedly through the axes.

2.1.1 Bintrees

Bintree partitions are simple in that every node is an interval and is split in half along one axis to give its children. They promote flexible and efficient use and access because each level of the space tree is a uniform grid of intervals. These grids are the most common partitions of the various function representations, yet allow neighborhoods with general shapes to be formed by taking the union of grid intervals from various tree levels.

The definition of a bintree can be based on the dyadic rationals

$$\left\{ i/2^\ell \mid i, \ell \in \mathcal{Z} \right\}$$

A hierarchy of one-dimensional intervals may be indexed by *level* ℓ and *position* i, as shown in Figure 2.1.1 for the subintervals of $[0, 1)$. These intervals are defined concisely as

$$I_{\ell,i} = \left[i/2^\ell, (i+1)/2^\ell \right)$$

The partition index is easily determined as $p(\ell, t) = \left\lfloor 2^\ell t \right\rfloor$.

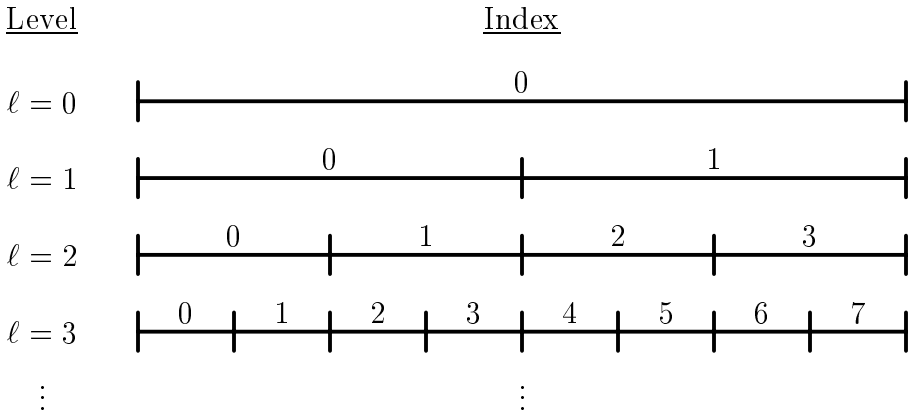


Figure 2.1.1: dyadic interval hierarchy

For higher dimensions, the hierarchy of one-dimensional intervals becomes a hierarchy of two- or three-dimensional intervals by splitting intervals in half along one axis at a time, as shown in Figure 2.1.2.

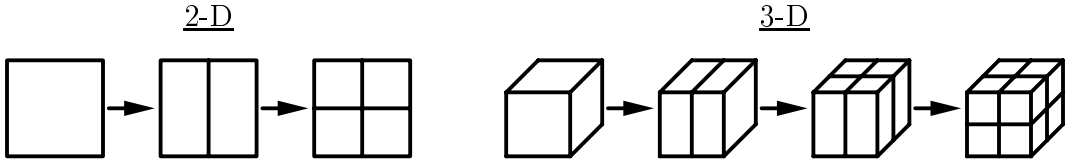


Figure 2.1.2: higher-dimensional interval hierarchies

The intervals here now have a level ℓ, current axis a, and n indices $i_1, \ldots, i_n$:

$$I_{\ell,a,i_1,\ldots,i_n} = I_{\ell+1,i_1} \times \cdots \times I_{\ell+1,i_{a-1}} \times I_{\ell,i_a} \times \cdots \times I_{\ell,i_n}$$

Partition indices can be determined seperately for each axis.

2.2 B-splines

B-splines are functions formed as piecewise polynomials with specific basis functions that facilitate analysis and refinement of the "pieces." For multivariate functions, the formulation considered here is the tensor-product form. B-splines have been found to be useful in a wide variety of modeling and rendering problems [2, 15].

The most elegant rigorous treatment of B-splines and refinement is through the use of symmetric n-affine functions called *blossoms* [32]. Using this general theory, along with a special theory introduced for dyadic refinement, several results are described in this section that are essential to the definition and application of dyadic splines in subsequent chapters.

Section 2.2.1 gives the univariate blossom formulation for B-splines with general (nonuniform) "pieces." Definitions are given for *knots*, *suites*, *blossoms* and *control points*. The extraction of piecewise polynomial B-spline functions from the blossom formulation is described.

Section 2.2.2 describes the blossom *pyramid diagram* used as a schematic for evaluation and refinement algorithms. This diagram also makes clear certain properties of B-splines that will be needed for efficient evaluation and bounding of B-spline functions.

The special case of dyadic refinement is central to this dissertation, and warrants examination from two viewpoints. The blossoming viewpoint (Section 2.2.3) is used to obtain concisely a weighted-average algorithm for performing dyadic refinement. As an alternative viewpoint, linear operators are used in Section 2.2.4 to obtain the refinement algorithm. These linear operators give a clear, concise way to express refinement, and will be valuable in introducing a set of analysis and synthesis operators in Chapter 4.

Least-squares fitting of B-splines to general target functions is discussed in section 2.2.5. This notion is critical for understanding hierarchical B-spline fitting (section 2.3), and is related to various fitting, prediction and approximation processes discussed in Chapters 4 and 5.

The discussion of B-splines concludes with a straightforward extension to multiple variables in section 2.2.6.

2.2.1 Blossoms and B-splines

A sequence of blossoms is constructed for a degree n and *knot sequence*

$$\cdots t_{-2} \leq t_{-1} \leq t_0 \leq t_1 \leq t_2 \cdots$$

The blossoms are denoted $f_i(x_1, \ldots, x_n)$, and are associated with the domain $[t_i, t_{i+1}]$.

If the interval is degenerate (i.e. $t_i = t_{i+1}$), no blossom is generated. As shown in Figure 2.2.1, each set of n consecutive knots is grouped into a *suite* which will be used as arguments to the blossoms at the *control points* for the B-spline.

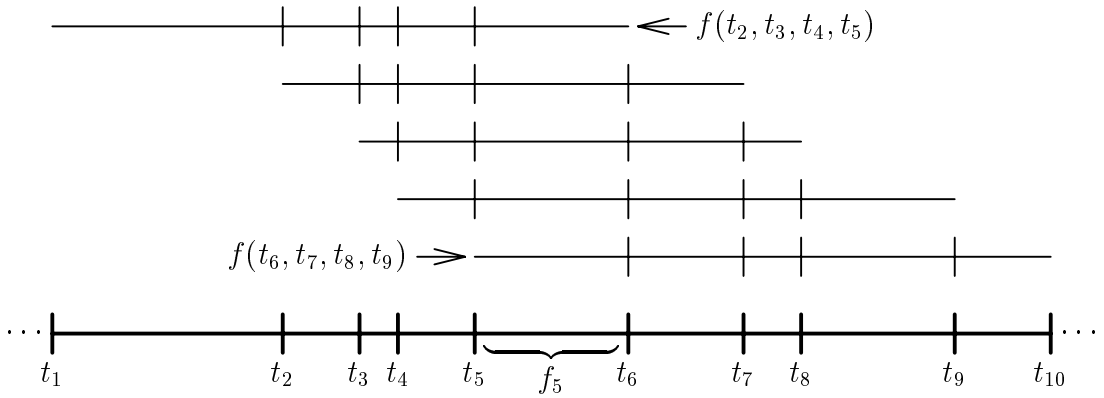


Figure 2.2.1: blossom control suites ($n = 4$)

Each blossom is uniquely determined by fixing its values at each of the $n + 1$ control-point suites that touch its associated domain:

$$\begin{gathered} f_i(t_{i-n+1}, \ldots, t_i) \\ f_i(t_{i-n+2}, \ldots, t_{i+1}) \\ \vdots \\ f_i(t_{i+1}, \ldots, t_{i+n}) \end{gathered}$$

The nth-degree polynomial function $F_i(t)$ for domain piece $[t_i, t_{i+1}]$ is then extracted from the *diagonal* of blossom f_i:

$$F_i(t) = f_i(t, t, \ldots, t)$$

By insisting that $f_i(t_{k1}, \ldots, t_{kn}) = f_j(t_{k1}, \ldots, t_{kn})$ when the two blossoms have the control suite in common, then the definition of the B-spline (and its unique associated blossoms) is complete (in fact, one may simply say $f(t_{k1}, \ldots, t_{kn})$ without ambiguity in this case).

This section has given a very short review of a rich subject. See Lyle Ramshaw's extensive technical report [32], in which the term *blossoming* is introduced. Independent work was introduced by de Casteljau [10]. An excellent introduction to this material is found in [11].

2.2.2 Pyramid Diagram

All properties and algorithms for B-splines pivot around the *pyramid diagram* for a given blossom f:

$$\begin{matrix} f(1,2,3) \\ f(2,3,4) \\ f(3,4,5) \\ f(4,5,6) \end{matrix} \searrow \begin{matrix} f(x_1,2,3) \\ f(x_1,3,4) \\ f(x_1,4,5) \end{matrix} \searrow \begin{matrix} f(x_1,x_2,3) \\ f(x_1,x_2,4) \end{matrix} \searrow f(x_1,x_2,x_3)$$

This is a generalization of the *de Boor algorithm* [9] and the *de Casteljau algorithm* [10]. The case $n = 3$ with $t_i = i$ is shown for domain piece $[3, 4]$.

One combination step diagram in the general case is

$$\begin{matrix} f(x_1,\ldots,x_k,t_j,\ldots,t_{j+n-k-1}) \\ f(x_1,\ldots,x_k,t_{j+1},\ldots,t_{j+n-k}) \end{matrix} \searrow f(x_1,\ldots,x_{k+1},t_{j+1},\ldots,t_{j+n-k-1})$$

for $k = 0, \ldots, n-1$. This diagram is shorthand for the equation

$$\begin{aligned} f(x_1,\ldots,x_{k+1},t_{j+1},\ldots,t_{j+n-k-1}) \quad &= \quad (1-u)\ f(x_1,\ldots,x_k,t_j,\ldots,t_{j+n-k-1}) + \\ &\qquad u\ f(x_1,\ldots,x_k,t_{j+1},\ldots,t_{j+n-k}) \end{aligned}$$

where

$$u = \frac{x_{k+1} - t_j}{t_{j+n-k} - t_j}$$

All of the properties of B-splines can be readily deduced from the formulation above. For example, the value $F_i(t) = f_i(t,\ldots,t)$ is a convex combination of the control points touching $[t_i, t_{i+1}]$. Another property is that when $t_{i-1} < t_i < t_{i+1}$, then $F_{i-1}(t)$ and $F_i(t)$ meet at t_i in their first $n-1$ derivatives.

The most important tool that falls out of the above formulation is a means by which control points can be obtained when new knots are added to the domain partition. The new control points are simply blossom evaluations at the new control point suites. This viewpoint is simpler that the original development of general B-spline refinement [7].

2.2.3 Dyadic Refinement

In the development of the dyadic splines, the cases of interest are when partitions are made up of successive levels of dyadic rationals, that is, going from a partition $t_{\ell,i} = i/2^\ell$ to $t_{\ell+1,i} = i/2^{\ell+1}$. Let the control points for the $t_{\ell,i}$ partition be

$$P_{\ell,i} = f(t_{\ell,i-\lfloor (n-1)/2 \rfloor},\ldots,t_{\ell,i+\lfloor n/2 \rfloor})$$

From the pyramid diagram it can be seen that $P_{\ell+1,2i}$ and $P_{\ell+1,2i+1}$ are weighted averages of $P_{\ell,i+j}$ for $j = -\lfloor (n+3)/4 \rfloor, \ldots, \lfloor (n+2)/4 \rfloor$. In fact, there are precisely two cases for each n:

$$\begin{aligned} P_{\ell,2i} &= \textstyle\sum_j \alpha_{n,j} P_{\ell-1,i+j} \\ P_{\ell,2i+1} &= \textstyle\sum_j \beta_{n,j} P_{\ell-1,i+j} \end{aligned}$$

where the $\alpha_{n,j}$ and $\beta_{n,j}$ weights are given in Figure 2.2.2a. A diagram of the domain intervals involved is given in Figure 2.2.2b.

Degree		$j=-1$	$j=0$	$j=1$
$n=1$	α	0	2/2	0
	β	0	1/2	1/2
$n=2$	α	1/4	3/4	0
	β	0	3/4	1/4
$n=3$	α	1/8	6/8	1/8
	β	0	4/8	4/8
$n=4$	α	5/16	10/16	1/16
	β	1/16	10/16	5/16

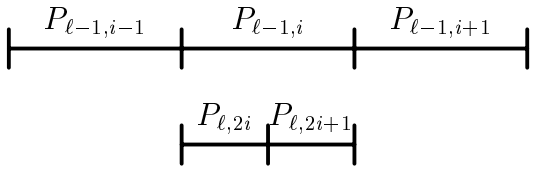


Figure 2.2.2: Dyadic Refinement
(a) weights (b) intervals

2.2.4 Alternative Derivation of Dyadic Refinement

A simple view of dyadic B-spline refinement is obtained through the application of linear operators on sequences of control points (see [12]). The subdivision operators obtained in this section play a central role in the definition of dyadic splines in Chapter 3. Similar operators are used throughout this dissertation.

Assume that a function is represented at a fixed *level* of resolution ℓ by a sequence of points $P_{\ell,i}$ indexed by $i = \ldots, -2, -1, 0, 1, 2, \ldots$. These point sequences will be referred to as *control-point vectors*. An example of a control-point vector is graphed in Figure 2.2.3a. The domain spacing between the graphed points is $1/2^\ell$. The goal is to repeatedly double the control-point vectors, that is, halve the domain spacing, while smoothing out the function in some reasonable way. The example control-point vector P_ℓ from Figure 2.2.3a is doubled in part (b), giving $P_{\ell+1}$ with domain spacing $1/2^{\ell+1}$. Starting at level $\ell = 0$, this gives a sequence of control-point vectors, P_ℓ for $\ell = 0, 1, 2, \ldots$, that hopefully converge to a smooth limit function.

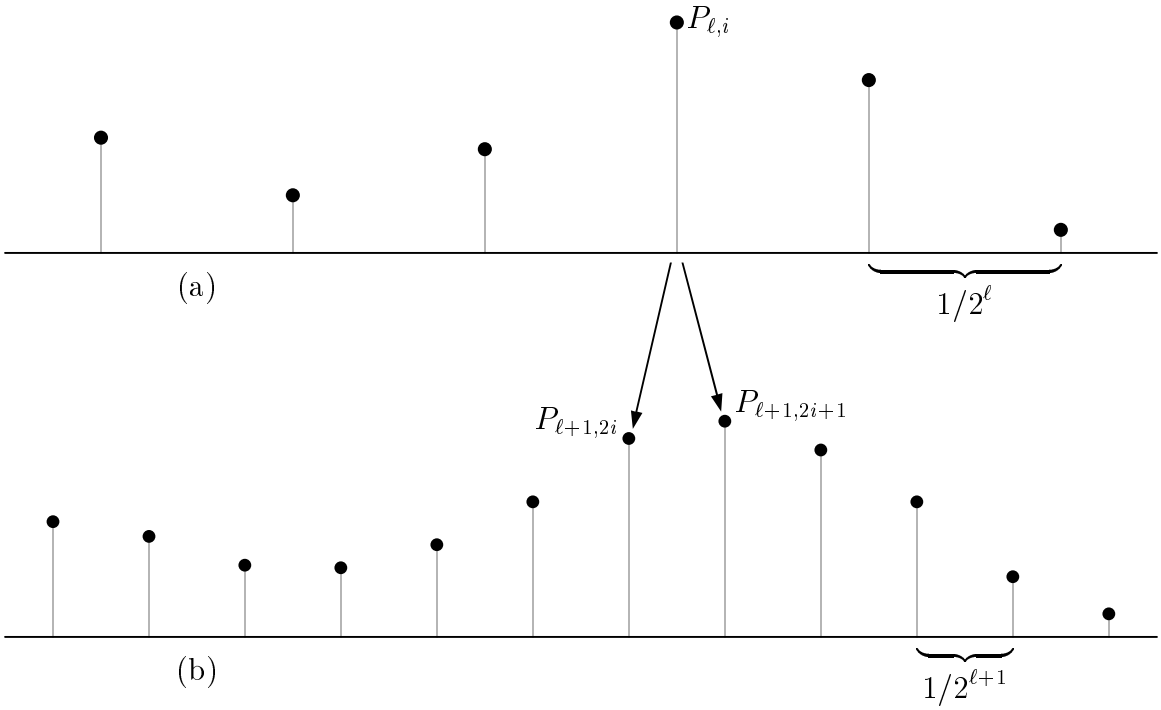


Figure 2.2.3: control-point vectors
(a) original (b) doubled

Before explaining how to double a control point vector, the general notion of applying linear operators to point sequences must be given. Let Q be a point sequence with elements Q_j for $j = \ldots, -2, -1, 0, 1, 2, \ldots$. A linear operator $\mathbf{A}$ on such sequences has elements $\mathbf{A}_{i,j}$ for $i, j \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Applying operator $\mathbf{A}$ to point sequence Q gives a new point sequence

$$R = \mathbf{A}Q$$

with elements

$$R_i = \sum_j \mathbf{A}_{i,j} Q_j$$

The simplest way to double a control-point vector is to make two copies of each point. Let the linear operator that does this be called the *split* operator $\mathbf{S}$. The elements of $\mathbf{S}$ can be defined as

$$\mathbf{S}_{i,j} = \begin{cases} 1 & \text{if } j = \lfloor i/2 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

For illustration, this simple split operator may be displayed as a section of an infinite

matrix:

$$\mathbf{S}_{i,j} = \begin{bmatrix} & & \vdots & & \\ & 1 & 0 & 0 & \\ & 1 & 0 & 0 & \\ \cdots & 0 & \underline{1} & 0 & \cdots \\ & 0 & 1 & 0 & \\ & 0 & 0 & 1 & \\ & 0 & 0 & 1 & \\ & & \vdots & & \end{bmatrix}$$

Note that the origin of the infinite matrix ($i = j = 0$) has been underlined. Applying $\mathbf{S}$ to perform the split on a control-point vector is expressed as

$$P_{\ell+1} = \mathbf{S}P_\ell$$

The limit as level ℓ increases is a step function. An example of this is shown in Figure 2.2.4.

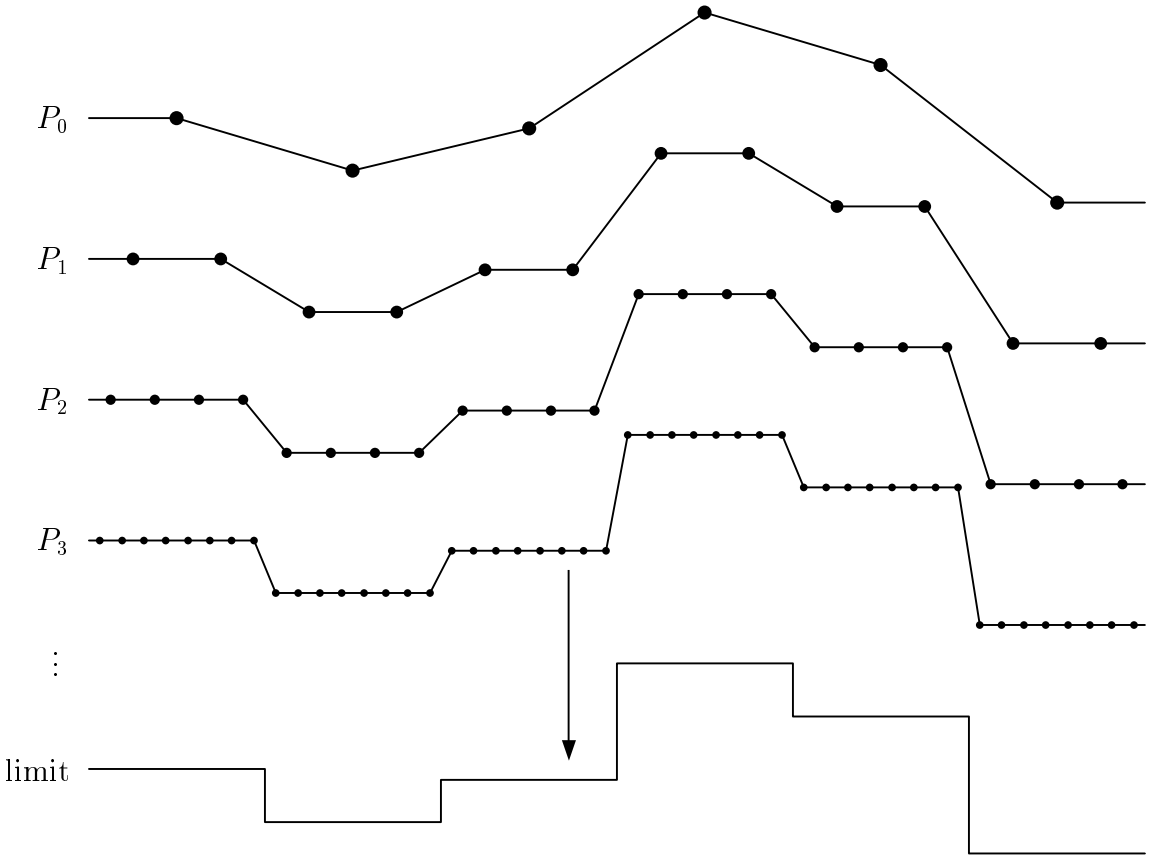


Figure 2.2.4: simple splitting gives a step function

To obtain a smoother limit function, doubling operators will be devised that act indirectly like the simple split operator. Instead of directly splitting the control-point vectors to obtain a piecewise constant limit function, the new doubling operators will cause the nth *differences* of control-point vectors to undergo simple splitting. Because a properly defined nth difference has the nth derivative as its limit, the new doubling operators will produce piecewise polynomials. The discussion that follows will define nth differences based on a difference operator, and then will define the new doubling operators that cause simple splitting of the nth difference.

The elements of difference operator $\mathbf{D}$ are defined as

$$\mathbf{D}_{i,j} = \begin{cases} -1 & \text{if } j = i \\ 1 & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

This has the inverse $\mathbf{D}^{-1}$ where

$$\mathbf{D}^{-1}_{i,j} = \begin{cases} 1 & \text{if } j < i \\ 0 & \text{otherwise} \end{cases}$$

These operators are displayed as

$$\mathbf{D}_{i,j} = \begin{bmatrix} & & \vdots & & & \\ & -1 & 1 & 0 & 0 & \\ \cdots & 0 & \underline{-1} & 1 & 0 & \cdots \\ & 0 & 0 & -1 & 1 & \\ & & \vdots & & & \end{bmatrix}$$

and

$$\mathbf{D}^{-1}_{i,j} = \begin{bmatrix} & & \vdots & & & \\ & 1 & 0 & 0 & 0 & \\ \cdots & 1 & 1 & \underline{0} & 0 & \cdots \\ & 1 & 1 & 1 & 0 & \\ & & \vdots & & & \end{bmatrix}$$

The sequence of nth differences for a sequence of control-point vectors P_ℓ is $2^{n\ell}\mathbf{D}^n P_\ell$. The factor $2^{n\ell}$ is needed since the domain spacing is cut in half for each increase in level. For the sequence of control-point vectors P_ℓ, the nth derivative of the limit function corresponds to the limit of the nth differences.

Simple splitting of control-point vectors produces a step function at the limit. A new doubling operation on P_ℓ will cause the points of the nth difference $2^{n\ell}\mathbf{D}^n P_\ell$ to undergo simple splitting. With this property of the new doubling operation, the limit function's nth derivative will be a step function. This implies that the limit function will be a piecewise nth-degree polynomial.

Let the nth-degree *smoothing* operator be defined as

$$\mathbf{M}^{(n)} = 1/2^n \mathbf{D}^{-n} \mathbf{S} \mathbf{D}^n$$

It remains to show that applying this operation as

$$P_{\ell+1} = \mathbf{M}^{(n)} P_\ell \tag{2.2.1}$$

causes simple splitting of the nth difference. This can be seen from the following:

$$\begin{aligned} 2^{n(\ell+1)} \mathbf{D}^n P_{\ell+1} &= 2^{n(\ell+1)} \mathbf{D}^n \mathbf{M}^{(n)} P_\ell \\ &= 2^{n(\ell+1)} \mathbf{D}^n (1/2^n \mathbf{D}^{-n} \mathbf{S} \mathbf{D}^n) P_\ell \\ &= \mathbf{S} (2^{n\ell} \mathbf{D}^n P_\ell) \end{aligned}$$

How does this relate to dyadic refinement of B-splines? By multiplying the infinite matrices together, the rows of the smoothing operator $\mathbf{M}^{(n)}$ are seen to be the dyadic refinement weights in Figure 2.2.2 (with a shift in indices). The columns of $\mathbf{M}^{(n)}$ are scaled and translated copies of row $n+1$ of Pascal's triangle

$$2^n \mathbf{M}^{(n)}_{i,j} = \binom{n+1}{2j+1-i} = \frac{(n+1)!}{(n+i-2j)!(2j+1-i)!}$$

This can be demonstrated by induction on n. For $n = 0$, this is true by inspection. Assuming $2^n \mathbf{M}^{(n)}_{i,j} = \binom{n+1}{2j+1-i}$, observe that

$$\begin{aligned} 2^{n+1} \mathbf{M}^{(n+1)}_{i,j} &= 2^n (\mathbf{D}^{-1} \mathbf{M}^{(n)} \mathbf{D})_{i,j} \\ &= 2^n \textstyle\sum_{u,v \in \mathcal{Z}} \mathbf{D}^{-1}_{i,u} \mathbf{M}^{(n)}_{u,v} \mathbf{D}_{v,j} \\ &= \textstyle\sum_{u<i} (2^n \mathbf{M}^{(n)}_{u,j-1} - 2^n \mathbf{M}^{(n)}_{u,j}) \\ &= \textstyle\sum_{u<i} \left(\binom{n+1}{2(j-1)+1-u} - \binom{n+1}{2j+1-u} \right) \\ &= \textstyle\sum_{w>2j-i} \left(\binom{n+1}{w-1} - \binom{n+1}{w+1} \right) \\ &= \binom{n+1}{2j-i} + \binom{n+1}{2j+1-i} \\ &= \binom{n+2}{2j+1-i} \end{aligned}$$

The last step is Pascal's triangle recurrence. The cases $n = 1, 2, 3$ are depicted as follows:

$$\frac{1}{2}\begin{bmatrix} & \vdots & \\ & 0\ \ 0\ \ 0 & \\ & 1\ \ 0\ \ 0 & \\ & 2\ \ 0\ \ 0 & \\ & 1\ \ 1\ \ 0 & \\ \cdots & 0\ \ \underline{2}\ \ 0 & \cdots \\ & 0\ \ 1\ \ 1 & \\ & 0\ \ 0\ \ 2 & \\ & 0\ \ 0\ \ 1 & \\ & \vdots & \end{bmatrix} \quad \frac{1}{4}\begin{bmatrix} & \vdots & \\ & 1\ \ 0\ \ 0 & \\ & 3\ \ 0\ \ 0 & \\ & 3\ \ 1\ \ 0 & \\ & 1\ \ 3\ \ 0 & \\ \cdots & 0\ \ \underline{3}\ \ 1 & \cdots \\ & 0\ \ 1\ \ 3 & \\ & 0\ \ 0\ \ 3 & \\ & 0\ \ 0\ \ 1 & \\ & \vdots & \end{bmatrix} \quad \frac{1}{8}\begin{bmatrix} & \vdots & \\ & 1\ \ 0\ \ 0 & \\ & 4\ \ 0\ \ 0 & \\ & 6\ \ 1\ \ 0 & \\ & 4\ \ 4\ \ 0 & \\ \cdots & 1\ \ \underline{6}\ \ 1 & \cdots \\ & 0\ \ 4\ \ 4 & \\ & 0\ \ 1\ \ 6 & \\ & 0\ \ 0\ \ 4 & \\ & \vdots & \end{bmatrix}$$

Figure 2.2.5a shows an example of the application of the piecewise-linear subdivision operator $\mathbf{M}^{(1)} = 1/2\mathbf{D}^{-1}\mathbf{S}\mathbf{D}$. Note that the limit function interpolates the initial control points. Part (b) of the figure shows the corresponding sequence of first differences and the limit derivative, which is a step function.

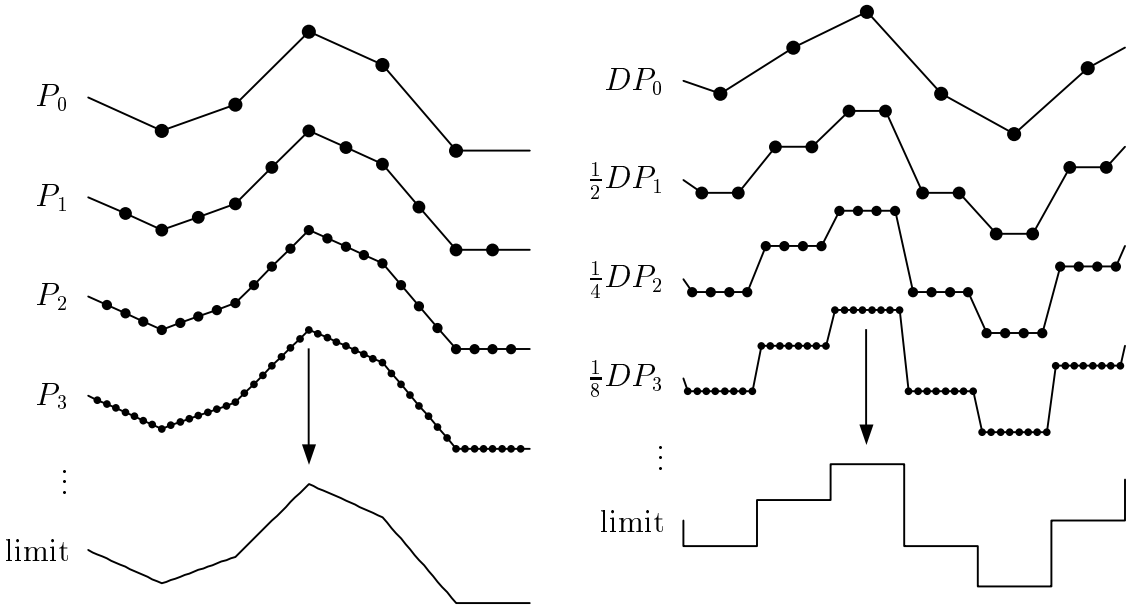


Figure 2.2.5: piecewise-linear subdivision
(a) control-point vectors (b) first differences

Figure 2.2.6 shows an example of the application of the piecewise-quadratic subdivision operator $\mathbf{M}^{(2)} = 1/4\mathbf{D}^{-2}\mathbf{S}\mathbf{D}^2$. Note that the limit function no longer interpolates the initial control points, but instead creates a smooth function that approximates the control points. Part (b) shows that the second differences converge to a step function—the second derivative of the limit function.

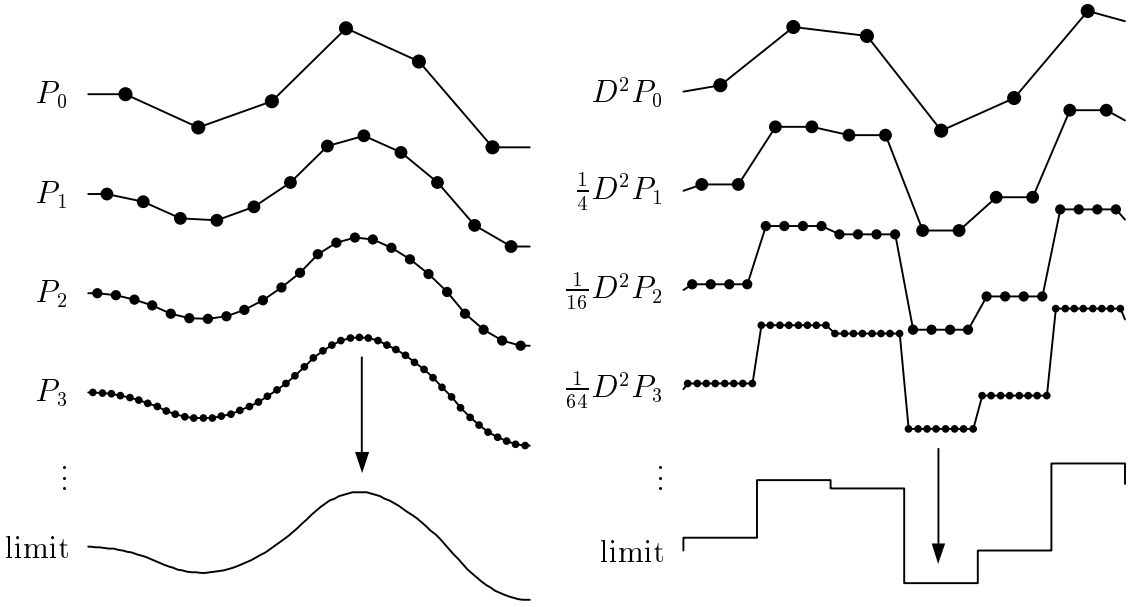


Figure 2.2.6: piecewise-quadratic subdivision
(a) control-point vectors (b) second difference

2.2.5 B-spline Least-Squares Fitting

Since B-spline functions depend linearly on their control points, it is straightforward to reduce least-squares fitting to matrix factorization.

Any B-spline function can be written as

$$f(x) = \sum_i P_i B_i(x)$$

where P_i are the control points and B_i are the basis functions. This can be written in vector form as

$$f = P^T B$$

Least-squares fitting is formulated for some target function $g(x)$ as finding the f that minimizes

$$\|f - g\|_2 = \left(\int_{x \in [x_0, x_1]} (f(x) - g(x))^2 \right)^{1/2}$$

This notation can be shortened into operator form as

$$\|f - g\|_2 = ((f - g) \cdot (f - g))^{1/2}$$

and may be simplified by noticing that the norm is minimized whenever $(f-g)\cdot(f-g)$ is minimized.

The minimization may be converted to a matrix formulation as follows:

$$\begin{aligned}(f-g)\cdot(f-g) &= (P^TB-g)\cdot(P^TB-g)\\ &= (P^TB)\cdot(P^TB)-2(P^TB)\cdot g+g\cdot g\\ &= P^T(BB^T)P-2(gB^T)P+g\cdot g\\ &= P^THP-2Q^TP+C\end{aligned}$$

where matrix H is given by

$$H_{i,j}=B_i\cdot B_j=\int_{x\in[x_0,x_1]}B_i(x)B_j(x)$$

which is sparse. The vector Q is given by

$$Q_i=g\cdot B_i=\int_{x\in[x_0,x_1]}g(x)B_i(x)$$

and constant C is given by

$$C=g\cdot g=\int_{x\in[x_0,x_1]}g(x)^2$$

The minimum of P^THP-2Q^TP+C is then found by computing the zero of its derivative with respect to P, that is, find P such that

$$2HP-2Q=0$$

which is equivalent to solving the matrix equation

$$HP=Q$$

This matrix equation can be solved by well-known sparse matrix factorization algorithms, such as *band Cholesky* [21, p. 154].

2.2.6 Multivariate B-splines

There are two predominant means of admitting more than one variable to a B-spline function. The one pioneered by de Casteljau forms an m-variate function for simplicial (e.g. triangular) domain pieces [10]. In this case, the blossoming formulation remains largely intact and the pyramid diagram becomes a sequence of nested triangular arrays of points. In the second method, the univariate basis functions are multiplied together to form m-variate basis functions—the *tensor product* B-splines [15]. This latter formulation is the one examined in this thesis.

The tensor-product B-splines are formulated for m axial degrees $n_1, \ldots, n_m$ and m knot vectors $t_{1,i_1}, t_{2,i_2}, \ldots, t_{m,i_m}$, where $i_1, \ldots, i_m \in \mathcal{Z}$. Using the blossoming formulation, the blossoms are

$$f_{\boldsymbol{i}}(x_{1,1}, x_{1,2}, \ldots, x_{1,n_1}; x_{2,1}, x_{2,2}, \ldots, x_{2,n_2}; \ldots; x_{m,1}, x_{m,2}, \ldots, x_{m,n_m})$$

where $\boldsymbol{i} = (i_1, i_2, \ldots, i_m)$ and each group $(x_{j,1}, \ldots, x_{j,n_j})$ is now a suite as in the univariate case. Here the blossom remains affine in each of its arguments, but is only symmetric for interchanges amongst members of a single suite. The control points are then

$$f(t_{1,i_1}, \ldots, t_{1,i_1+n-1}; t_{2,i_2}, \ldots, t_{2,i_2+n-1}; \ldots; t_{m,i_m}, \ldots, t_{1,i_m+n-1})$$

and the multivariate function is extracted from the diagonal as

$$F(x_1, \ldots, x_m) = f(x_1, \ldots, x_1; x_2, \ldots, x_2; \ldots; x_m, \ldots, x_m)$$

Finally, the pyramid diagram can be constructed independently for any interchange suite $(x_{j,1}, \ldots, x_{j,n_j})$ when the other blossom arguments are fixed and equal.

For example, suppose $m = 2$, $n_1 = 2$ and $n_2 = 3$. In this case, with knot vectors $(\ldots, -2, -1, 0, 1, 2, \ldots)$ on both axes, the pyramid diagrams come out as follows:

$$\left.\begin{matrix} f(1,2;1,2,3) \\ f(2,3;1,2,3) \\ f(3,4;1,2,3) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,2;1,2,3) \\ f(x_1,3;1,2,3) \end{matrix}\right\rangle f(x_1,x_2;1,2,3)$$

$$\left.\begin{matrix} f(1,2;2,3,4) \\ f(2,3;2,3,4) \\ f(3,4;2,3,4) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,2;2,3,4) \\ f(x_1,3;2,3,4) \end{matrix}\right\rangle f(x_1,x_2;2,3,4)$$

$$\left.\begin{matrix} f(1,2;3,4,5) \\ f(2,3;3,4,5) \\ f(3,4;3,4,5) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,2;3,4,5) \\ f(x_1,3;3,4,5) \end{matrix}\right\rangle f(x_1,x_2;3,4,5)$$

$$\left.\begin{matrix} f(1,2;4,5,6) \\ f(2,3;4,5,6) \\ f(3,4;4,5,6) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,2;4,5,6) \\ f(x_1,3;4,5,6) \end{matrix}\right\rangle f(x_1,x_2;4,5,6)$$

$$\left.\begin{matrix} f(x_1,x_2;1,2,3) \\ f(x_1,x_2;2,3,4) \\ f(x_1,x_2;3,4,5) \\ f(x_1,x_2;4,5,6) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,x_2;y_1,2,3) \\ f(x_1,x_2;y_1,3,4) \\ f(x_1,x_2;y_1,4,5) \end{matrix}\right\rangle \left.\begin{matrix} f(x_1,x_2;y_1,y_2,3) \\ f(x_1,x_2;y_1,y_2,4) \end{matrix}\right\rangle f(x_1,x_2;y_1,y_2,y_3)$$

With the tensor-product blossoms, refinement may be performed on any combination of axis knot vectors. For the purposes of dyadic splines, dyadic refinement is performed on one axis at a time, corresponding to the partitions and split axes in a bintree. For this, the weighting scheme given earlier is applied along the split-axis direction. Thus the tensor product formulation works well with the bintree partitioning scheme.

2.3 Hierarchical B-splines

The idea of refining a B-spline domain has been around for many years. In the early schemes, detail was added by first applying a uniform refinement to the entire domain, and then modifying the control points. Forsey and Bartels [18] developed the concept of *hierarchical B-splines*, where offset detail could be added to a B-spline function as it is refined. This gave the designer the ability to retain coarse control of the function while adding detail through refinement.

Forsey and Bartels describe a further enhancement to the definition of the offsets so that they are specified with respect to local coordinate frames derived from the coarse, unperturbed parent surface. This has the desirable effect that the offset shape tracks any changes in the base surface in an intuitive way. A further enhancement along these lines was introduced in Barghiel's thesis [1], where more general offset placement and overlaps are considered.

2.3.1 Forsey-Bartels Offsets

Forsey and Bartels utilize B-spline refinement to convert coarse control points to a finer set. Detail is then introduced by specifying *offsets* to the finer set of control points. These offsets are added to the (unperturbed) fine set of control points to produce the final control points defining the function. The resulting function can be controlled at both coarse and fine levels of detail, where the detail control is relative to the coarse control. Finally, this process of refining and offsetting can be repeated, resulting in a hierarchy of control offsets at multiple levels of detail.

The Forsey-Bartels offsets are grouped into disjoint domain rectangles at each layer of refinement. Each offset rectangle from one layer must be a subset of an offset rectangle at the next coarser level. Each offset rectangle contains at its center the offset control points that are to be manipulated, and these are surrounded by a buffer of null offsets to preserve continuity. When additional points are to be offset, a rectangle is expanded to accommodate this. When two rectangles for a layer overlap, they are merged into one large rectangle.

Although the formulation of Forsey and Bartels provides intuitive multiresolution control, it suffers from two drawbacks. First, defining and maintaining the Forsey-Bartels offsets is complex, requiring offset rectangles to be refined and merged for all possible tensor-product knot partitions. Furthermore, very sparse collections of offset points can cause the Forsey-Bartels rectangles to degenerate into the entire domain. For example, consider offset points along a diagonal in the domain.

The dyadic spline formulation and evaluation mechanism will be different from that of Forsey and Bartels, but will preserve the primary desirable qualities of their method. These qualities are: multiresolution control, multiresolution fitting, and

reduction to B-spline pieces. Dyadic splines do not use the general knot placement available to Forsey and Bartels' hierarchical B-splines. Such general knot placements are advantageous in some applications. However, the multiresolution nature of dyadic splines implies that the effect of nonuniform knot placements can be achieved as accurately as desired by using finer details.

2.3.2 Hierarchical B-spline Fitting

With hierarchical B-splines, the goal of fitting is to create a hierarchy of offsets that cause a hierarchical B-spline to closely approximate a given function. This setting creates challenges and opportunities beyond the usual least-squares fitting of conventional B-splines. Since refinement can occur indefinitely, hierarchical B-splines can be fit to within any desired tolerance of a function, using pointwise error measurements (the supremum norm) in addition to least-squares error norms. The reason for using hierarchical B-splines over traditional B-splines is that the resulting representation can be much more compact (although the problems identified earlier with the Forsey-Bartels formulation will defeat this in significant cases).

The fitting algorithm described in Forsey's PhD thesis [17], and in a subsequent paper [19], is organized as follows. The algorithm proceeds top-down, using a stack of rectangular offset domains. For each offset domain, an error metric is stored, and only those rectangles whose error is above a certain tolerance are kept. Initially, the stack contains a single, coarse offset rectangle. The top of the stack is repeatedly popped, fit and analyzed. Fitting for a single offset rectangle is done using least-squares fitting, as described in the previous section but with sampled data. If the portion of the hierarchical B-spline associated with this offset rectangle is within tolerance, it is output. Otherwise the analysis consists of checking for disjoint subrectangles that cover all intolerance regions. If such intolerance subrectangles are found, they are pushed on the stack. If not, the region is refined uniformly and pushed back on the stack.

This fitting method takes advantage of the situation where isolated intolerance regions exist. In this way, sparse collections of offsets are formed. However, the structure of the Forsey-Bartels formulation prevents the algorithm from functioning well in the case of sparse but non-isolated intolerance regions. A further difficulty is that the search for intolerance regions involves using fill algorithms on a raster of the error. Dyadic splines will overcome these difficulties while retaining the desirable behavior.

2.4 Wavelet Multiresolution Analysis

The most useful wavelets in practice are defined through *multiresolution analysis*, as pioneered by Mallat [26]. In multiresolution analysis, a *scaling function* $\phi(t)$ is

dilated and translated to form basis functions for a nested sequence of function spaces

$$\cdots V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \cdots$$

Typically, $V_{\ell+1}$ is made twice the resolution of V_ℓ. The basis for V_ℓ is

$$\{\phi_{\ell,i} \mid i \in \mathcal{Z}\}$$

where

$$\phi_{\ell,i}(t) = \phi(2^\ell t - i)$$

The compliment spaces W_ℓ are defined as

$$V_{\ell+1} = V_\ell + W_\ell$$

and *wavelets*

$$\psi_{\ell,i}(t) = \psi(2^\ell t - i)$$

form bases for W_ℓ for an appropriate choice of ψ. The wavelets $\psi_{\ell,i}$ are derived so that they form bases for the differences in functions stored at successive resolutions (in effect, the wavelets at resolution $\ell + 1$ restore detail that can't be maintained at the coarser resolution ℓ).

Multiresolution analysis consists of a sequence of *filter banks* to *decompose* and *reconstruct* a function f. To understand filter banks, one fact to note is that the scaling and wavelet functions at resolution level ℓ, $\phi_{\ell,i} \in V_\ell$ and $\psi_{\ell,i} \in W_\ell$, can be expressed as linear combinations of scaling functions $\phi_{\ell+1,i}$ from level $\ell + 1$ (this follows immediately from the fact that $V_\ell \subset V_{\ell+1}$ and $W_\ell \subset V_{\ell+1}$). Let these linear relationships be denoted

$$\phi_{\ell,i} = \sum_j \alpha_j \phi_{\ell+1,2i+j}$$

and

$$\psi_{\ell,i} = \sum_j \beta_j \phi_{\ell+1,2i+j}$$

The weights α and β form the columns of the reconstruction operators $\mathbf{M}$ and $\mathbf{E}$:

$$\mathbf{M}_{i,j} = \alpha_{j-2i}$$

and

$$\mathbf{E}_{i,j} = \beta_{j-2i}$$

Reconstruction (i.e. synthesis) is defined as follows. Let $f_\ell = P_\ell \cdot \phi_\ell$ and $g_\ell = Q_\ell \cdot \psi_\ell$. Then

$$P_{\ell+1} = \mathbf{M}P_\ell + \mathbf{E}Q_\ell$$

is the reconstruction step in a filter bank. The operator $\mathbf{M}$ is the *refinement* or *subdivision* operator, while the $\mathbf{E}$ operator *expands* the missing detail. This can be expressed as a single operator:

$$P_{\ell+1} = [\mathbf{M}|\mathbf{E}] \begin{bmatrix} P_\ell \\ Q_\ell \end{bmatrix}$$

The decomposition (i.e. analysis) step is then formally defined as

$$\begin{bmatrix} P_\ell \\ Q_\ell \end{bmatrix} = [\mathbf{M}|\mathbf{E}]^{-1} P_{\ell+1}$$

The decomposition operator is split into a *fit* operator $\mathbf{F}$ and a *compaction of difference* operator $\mathbf{C}$:

$$\begin{bmatrix} \mathbf{F} \\ \mathbf{C} \end{bmatrix} = [\mathbf{M}|\mathbf{E}]^{-1}$$

The decomposition step can now be expressed in two parts,

$$P_\ell = \mathbf{F} P_{\ell+1}$$

and

$$Q_\ell = \mathbf{C} P_{\ell+1}$$

A diagram of the decomposition and reconstruction cycle is shown in Figure 2.4.1.

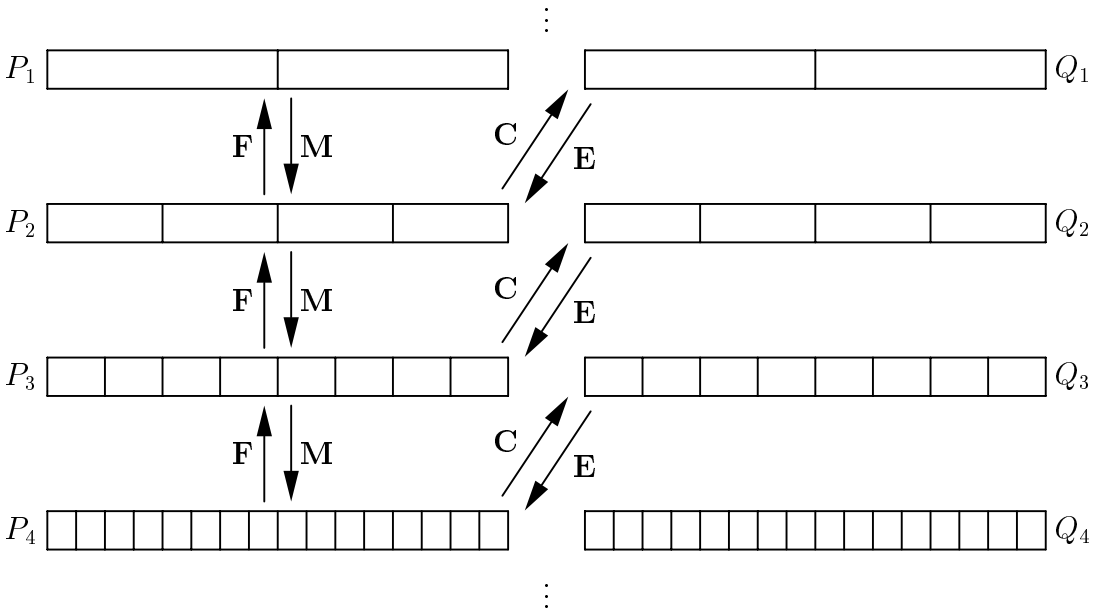


Figure 2.4.1: wavelet filter bank

Mallat's original paper uses orthonormal scaling function bases $\phi_{\ell,i}$, but the technique also works for nonorthogonal scaling function bases such as those associated with uniform B-splines[1] under dyadic refinement [6, 35]. The multiresolution basis functions for cubic uniform B-splines are shown in Figure 2.4.2.

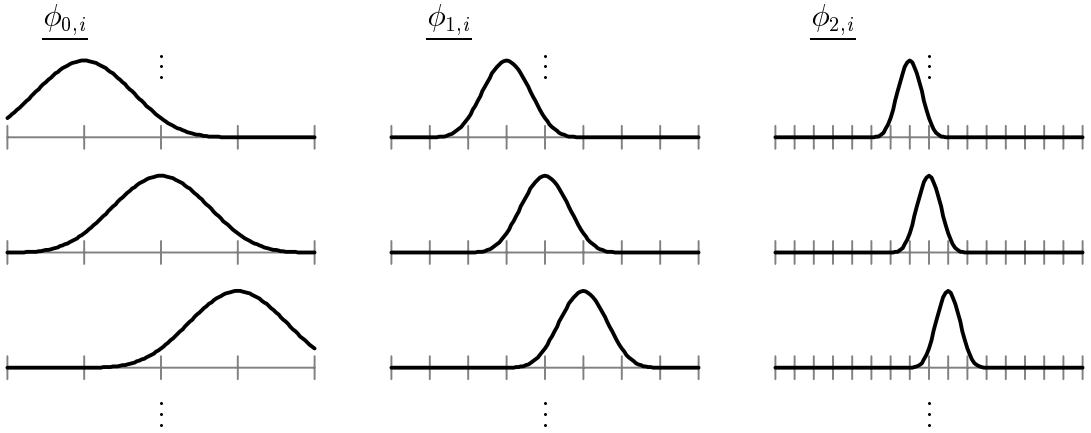


Figure 2.4.2: $\phi_{\ell,i}$ for cubic dyadic refinement

Many different wavelets ψ may be chosen to compliment the scaling functions ϕ. Figure 2.4.3 depicts cubic wavelets of minimal support (sometimes called *B-wavelets*) [6].

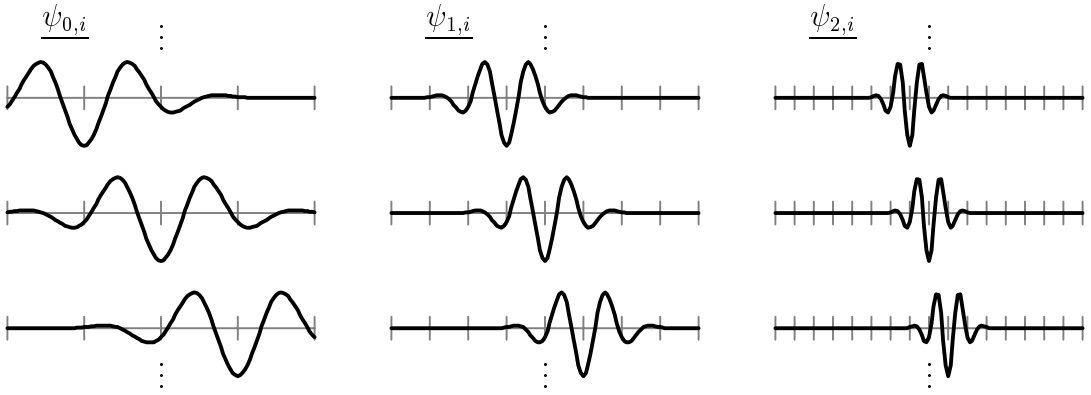


Figure 2.4.3: example $\psi_{\ell,i}$ for cubic dyadic refinement

Researchers in many fields are using wavelets to good effect. Examples include image compression [24], image understanding [26], multiresolution editing [3, 16], and

[1] Chui [6] refers to uniform B-splines as *cardinal splines*.

solution of diffuse global illumination [22]. Compression is a natural byproduct of the wavelet transform since coefficients of transformed functions tend to have large magnitudes concentrated at coarse resolutions or along edges. The bulk of the coefficients are small and can be eliminated or stored with a small number of bits. Image compression ratios have been reported that are are significantly better than JPEG for a range of signal-to-noise ratios.

Although wavelets have had many outstanding benefits in several fields (including graphics), several improvements are needed. First, the wavelet coefficients obtained after decomposition do not make good design handles, whereas B-spline control points do. Also, B-splines have more elegant refinement, bounding and differentiation properties. Wavelet decompositions tend to be preprocessed, yet detail should be produced exactly where it is needed. Furthermore, wavelet approximations are constructed bottom-up, and a top-down approximation method is often necessary. Finally, a simple family of exact finite-width decomposition and reconstruction filters are needed for smooth wavelets. Dyadic splines offer these improvements.

2.5 Fractals

The development of fractals has brought with it a host of mathematical and algorithmic tools for synthesizing the geometry of a wide variety of natural phenomena. A key technique is the use of functions of one, two or three variables that are "rough" in a sense that may be quantified through Fourier analysis, statistical measures of self-similarity, or a fractal dimension. Numerous algorithmic constructions have been proposed for such functions which vary widely in their simplicity, speed, data space, function quality, and flexibility of use. A survey of these construction methods may be found in [34], while a more general treatment of fractals is given in [27].

The simplest time- and space-efficient technique is that of midpoint displacement [20]. Unfortunately the functions produced in this way have creases where first-derivative discontinuities are introduced along grid lines. To solve this problem, slower, more complicated variants on midpoint displacement were later introduced [29, 28].

The univariate midpoint displacement construction is depicted in Figure 2.5.1. Initially, two endpoints are given at level $\ell = 0$ that define a line segment over an interval. This segment is split at its midpoint, which is then perturbed by an amount chosen at random from a given distribution (such as a uniform distribution over $[-1, 1]$). This yields the two-segment function at level $\ell = 1$. This process is repeated to produce subsequent levels, but with the expected size of the perturbation halved at each level. An example limit function is shown.

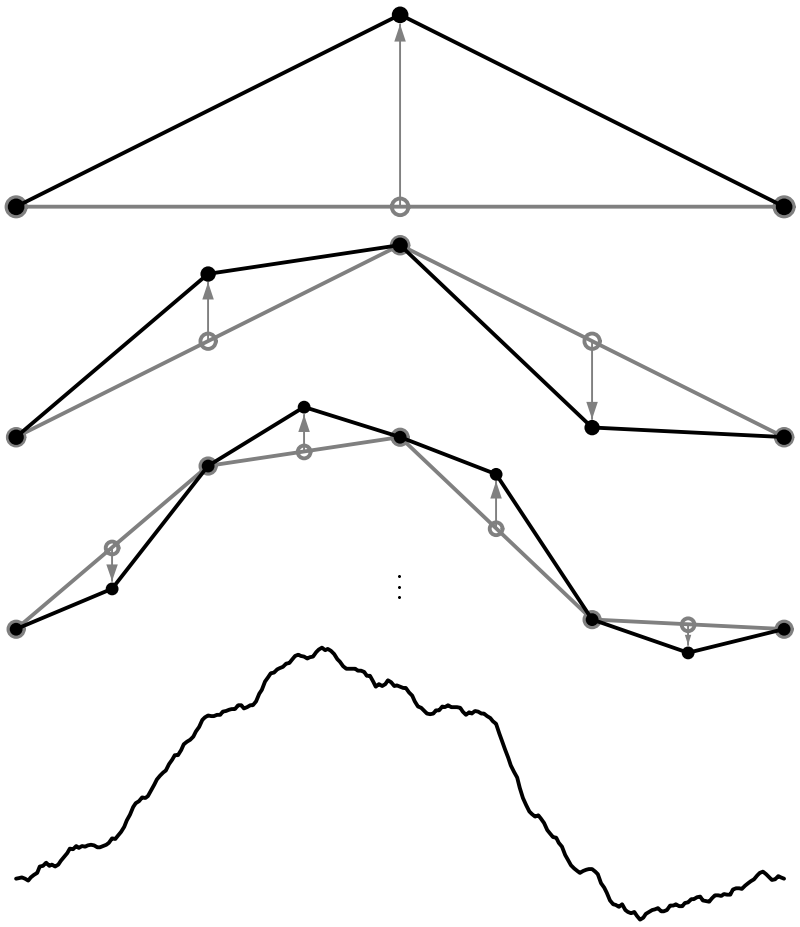


Figure 2.5.1: midpoint displacement

Random fractal functions have been used extensively to produce synthetic models of landscapes, clouds and mineral formations. From the point of view of geometric modeling and rendering, the primary advantage of fractals is that natural detail can be produced automatically to increase the realism of synthetic models and images.

The midpoint displacement construction carries with it many of the desirable qualities of B-spline recursive subdivision. First, the output may be produced to any level of detail just where it is needed, without preprocessing. In contrast, the methods for computing random fractals before [20] emphasized preprocessing at fixed, uniform resolutions. Second, midpoint displacement fractals can be recursively subdivided in the same manner as B-splines. This gives midpoint displacement the same opportunities that B-splines have for taking advantage of scale-optimization efficiencies, such as culling geometry that is outside the field of view at the coarsest possible resolution.

A disadvantage of midpoint displacement and the other traditional methods of producing random fractals is that the process is difficult to control to produce specific

features in the output geometry. This is perhaps the most critical factor in using fractal functions to design synthetic natural forms: having the ability to control the function shape in an intuitive way. The traditional fractal formulations do not facilitate editing of either the positions or the random-number processes.

Another disadvantage of traditional fractal constructions is that they are either strictly preprocessed (e.g. [27]) or strictly online (e.g. midpoint-displacement recursive subdivision). If the construction is preprocessed, the expensive function evaluations can be reused, but the function must then be computed where it is not needed or to insufficient accuracy. If the construction is online, then computation reuse is abandoned in favor of adaptability to runtime conditions.

Dyadic splines overcome these limitations. Because dyadic splines are closely related to B-splines and hierarchical B-splines, powerful editing mechanisms are available for dyadic-spline fractals. As discussed in section 3.3, evaluation of dyadic splines can simultaneously gain the advantages of preprocessing and online processing. Most important, dyadic splines allow fractal functions to be smoothly integrated into the diverse applications that use multiresolution functions.

Chapter 3

Function Synthesis

This chapter describes the formulation and evaluation mechanisms for dyadic splines of one variable. The general recurrence formula relating control displacements and range positions is given first. The technique developed here is to view the function domain as a simple hierarchy of intervals (the bintree), to provide control through displacements at each interval, and to compute resulting positions using the B-spline dyadic refinement weighting scheme. After giving the dyadic-spline recurrence formula, the problem of evaluation is discussed. First, it is observed that dyadic splines behave like B-splines locally, and can thus be sampled and bounded by recursively evaluating local B-splines. Next, based on this observation, specific implementation strategies and solutions are given.

3.1 Formulation

This section will define dyadic splines based on bintree domain intervals, displacements and positions. The dyadic-spline recurrence will be given that relates displacements and positions. Dyadic spline functions will be defined in terms of limits of the positions.

Recall from section 2.1 that the bintree intervals for a univariate domain are $I_{\ell,i} = \left[i/2^\ell, (i+1)/2^\ell\right)$. Displacements and range positions associated with $I_{\ell,i}$ will be denoted by $D_{\ell,i}$ and $P_{\ell,i}$ respectively. An interval is called *left* if i is even and *right* if i is odd. For simplicity, assume that initial positions are given at level $\ell = 0$.

The index relationships and domain intervals for a generic parent and its children are shown in Figure 3.1.1a. A tree of indices is shown in 3.1.1b for $t \in [0, 1]$.

The combined weighting and displacement scheme is written as the left and right recurrences of the general form

$$\begin{aligned} P_{\ell,2i} &= \textstyle\sum_j \alpha_{n,j} P_{\ell-1,i+j} + D_{\ell,2i} \\ P_{\ell,2i+1} &= \textstyle\sum_j \beta_{n,j} P_{\ell-1,i+j} + D_{\ell,2i+1} \end{aligned}$$

This is a generalization of the recurrence given earlier in section 2.2.3 for B-spline dyadic refinement, and the degree n weight masks $\alpha_{n,j}$ and $\beta_{n,j}$ remain the same. The application of these weight masks is depicted in Figure 3.1.2 for the cubic case. Note that the left and right child positions are both computed as weighted averages of a neighborhood of positions around their parent.

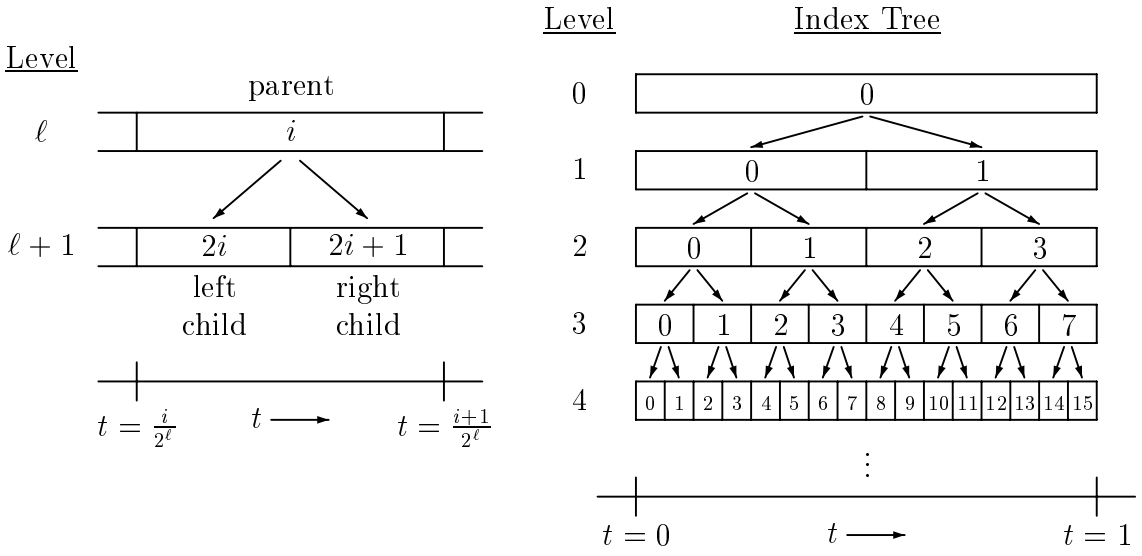


Figure 3.1.1: domain indices
(a) parent and children (b) indices for $t \in [0, 1]$

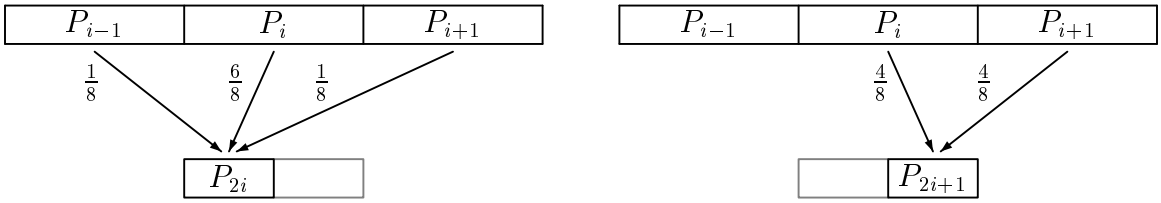
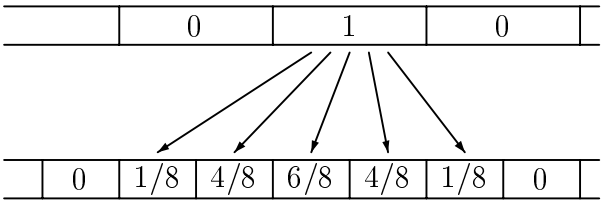


Figure 3.1.2: cubic weight masks
(a) left child (b) right child



Degree	Footprint	Scale
$n = 0$	$\underline{1}$ 1	1
$n = 1$	1 $\underline{2}$ 1	1/2
$n = 2$	1 $\underline{3}$ 3 1	1/4
$n = 3$	1 4 $\underline{6}$ 4 1	1/8
$n = 4$	1 5 $\underline{10}$ 10 5 1	1/16
	$\vdots$	

Figure 3.1.3: (a) cubic footprint (b) general footprints

Another view of the weighting scheme is obtained by examining the *footprint* of weights that a parent contributes to a neighborhood around its children. The footprint in the cubic case is shown in Figure 3.1.3a. The footprints for general degree n are shown in Figure 3.1.3b. Note that these footprints are scaled copies of the rows in Pascal's triangle, as demonstrated earlier in section 2.2.4. These footprints appear in the columns of the subdivision operators[2] $\mathbf{M}^{(n)}$:

$$\cdots \quad \mathbf{M}^{(2)} = \frac{1}{4}\begin{bmatrix} & \vdots & \\ 3 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & \underline{3} & 1 & 0 \\ \cdots\ 0 & 1 & 3 & 0\ \cdots \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 \\ & \vdots & \end{bmatrix} \quad \mathbf{M}^{(3)} = \frac{1}{8}\begin{bmatrix} & \vdots & \\ 4 & 0 & 0 & 0 \\ 6 & 1 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 1 & \underline{6} & 1 & 0 \\ \cdots\ 0 & 4 & 4 & 0\ \cdots \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 4 \\ & \vdots & \end{bmatrix} \quad \cdots$$

In operator notation, the *dyadic-spline recurrence* is

$$P_{\ell+1} = \mathbf{M}P_\ell + D_{\ell+1} \tag{3.1.1}$$

where $\mathbf{M} = \mathbf{M}^{(n)}$ for some $n \geq 0$.

A function $f(t)$ is now defined at any point t as the limit of the positions $P_{\ell,i}$ associated with intervals containing t. Any function $f(t)$ that is well defined by this limit process will be called a *dyadic spline*. Of course, such a limit does not exist without restricting the displacements in some way. In practice, a simple, slightly conservative restriction will be used. This restriction involves insisting that all but a finite number of displacements for f satisfy

$$|D_{\ell,i}| < \mu\tau^{-\ell} \tag{3.1.2}$$

for some constants $\mu > 0$ and $\tau > 1$. This restriction will apply to automatically generated displacements, such as those used to produce natural detail (e.g. fractals). The exceptions will be user edits of displacements, either direct or indirect.

[2]Note that the rows have been shifted down compared to the refinement operators derived in section 2.2.4, but are otherwise identical. This shift is used to center the basis functions around the bintree interval associated with its index. This shift is consistent with the indexing used earlier in section 2.2.3.

3.2 Sampling and Bounding

For robust evaluations of the limit function, it is vital to provide guaranteed bounds on the image of domain intervals, as well as for derivatives of the function. It is generally better to know how a function behaves over a whole interval rather than at a single point. Nevertheless, it is convenient to provide point-sample evaluations at corners of intervals in the domain hierarchy (e.g. for tessellations). Both corner samples and guaranteed bounds for interval images are readily computed from the range positions P and displacements D of the dyadic spline.

Consider the function shown in Figure 3.2.1, which has nonzero displacements only at level $\ell = 0$. The dyadic-spline formulation reduces to uniform B-spline subdivision in this case. To see this, note that without displacements, Equation 3.1.1 is identical to the operator form of uniform B-spline subdivision given in Equation 2.2.1. Therefore, this dyadic-spline function is a uniform B-spline with control points $P_{0,i} = D_{0,i}$. For uniform B-splines it is a simple matter to evaluate the function and its derivatives for arbitrary points t and for any interval $I_{\ell,i}$ [15].

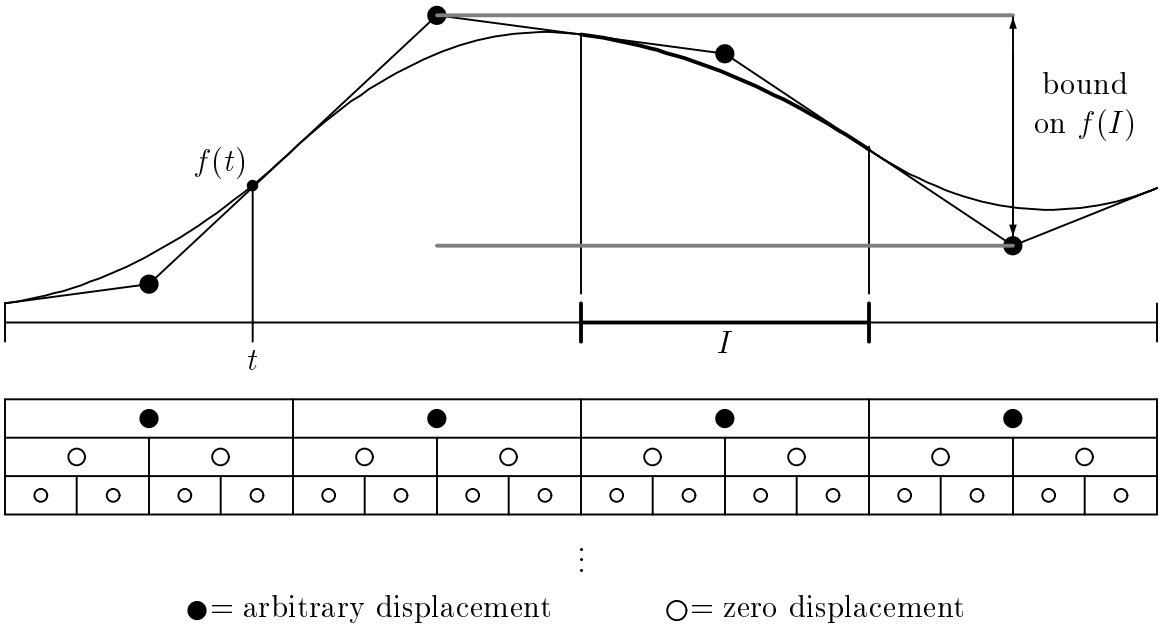


Figure 3.2.1: function with displacements at $\ell = 0$

For an interval $I_{\ell,i}$, upper and lower bounds on $f(t)$ for $t \in I_{\ell,i}$ can be quickly determined by taking the minimum and maximum $P_{\ell,i+j}$ for a small neighborhood around $I_{\ell,i}$, as shown in the figure. Note that these bounds tend to be loose, but are inexpensive to compute. The neighborhood of positions $P_{\ell,i+j}$ that influence $f(t)$ for

$t \in I_{\ell,i}$ are shown in Figure 3.2.2a. Looking at this another way, Figure 3.2.2b shows the intervals $I_{\ell,i+j}$ that are influenced by a range position $P_{\ell,i}$. Also notice that $D_{\ell,i}$ influences $I_{\ell,i+j}$ exactly when $P_{\ell,i}$ does.

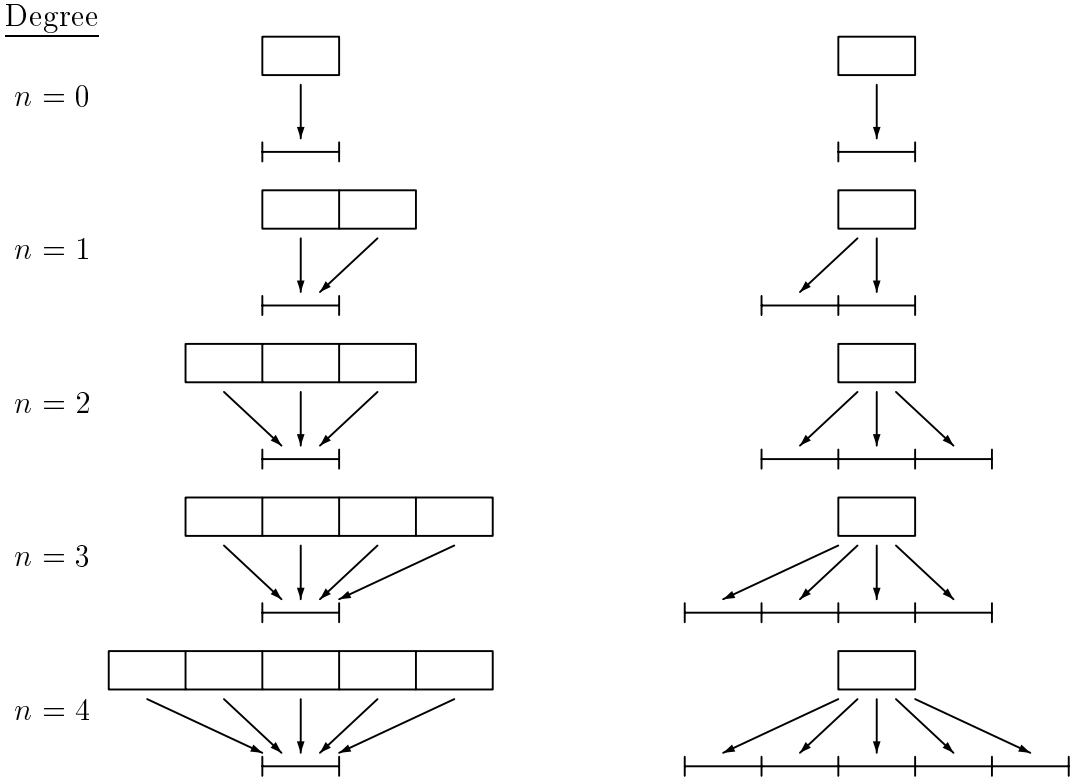


Figure 3.2.2: (a) $P_{\ell,i+j}$'s influencing an interval $I_{\ell,i}$
(b) intervals $I_{\ell,i+j}$ influenced by $P_{\ell,i}$

The observation that $f(t)$ is a B-spline also holds for any level ℓ when displacements $D_{\ell',i}$ are zero for lower levels $\ell' > \ell$. The points $P_{\ell,i}$ determine the exact limit function by acting as control points for a uniform B-spline, but now at a higher resolution.

The observation can be broadened further by noting that $f(t)$ will be a B-spline *locally* over an interval $I_{\ell,i}$ whenever the displacements that influence $I_{\ell,i}$ below level ℓ are zero. The displacements that must be zero are depicted for the cubic case in Figure 3.2.3. This set of displacements can be derived in the general case by applying Figure 3.2.2a to all the descendents of $I_{\ell,i}$.

With the observations so far, a dyadic-spline function $f:[0,1] \to \Re$ with a finite number of nonzero displacements can be converted to a finite collection of local B-splines. The conversion algorithm recursively descends the domain interval tree whenever $f(t)$ is not guaranteed to be a B-spline locally for $t \in I_{\ell,i}$. The algorithm stops

recursing when $f(t)$ is a B-spline over $I_{\ell,i}$. Figure 3.2.4 shows the local B-spline intervals that result for a cubic dyadic spline with three nonzero displacements. Corner samples are marked that delimit the B-spline pieces.

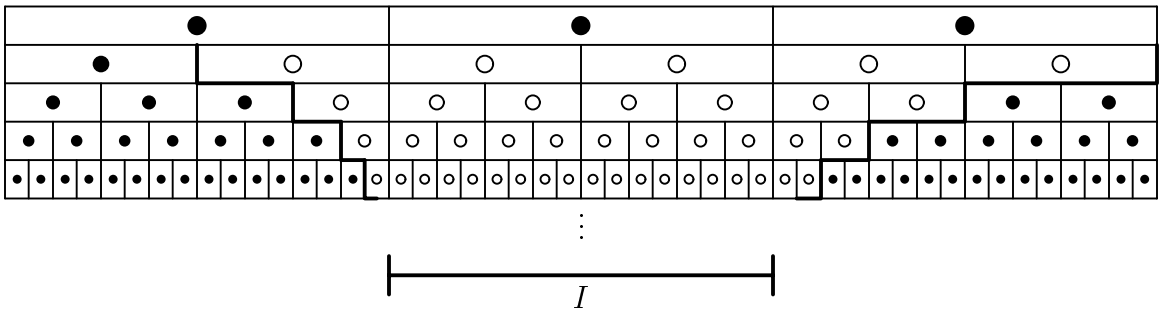


Figure 3.2.3: zero displacements for local cubic B-spline

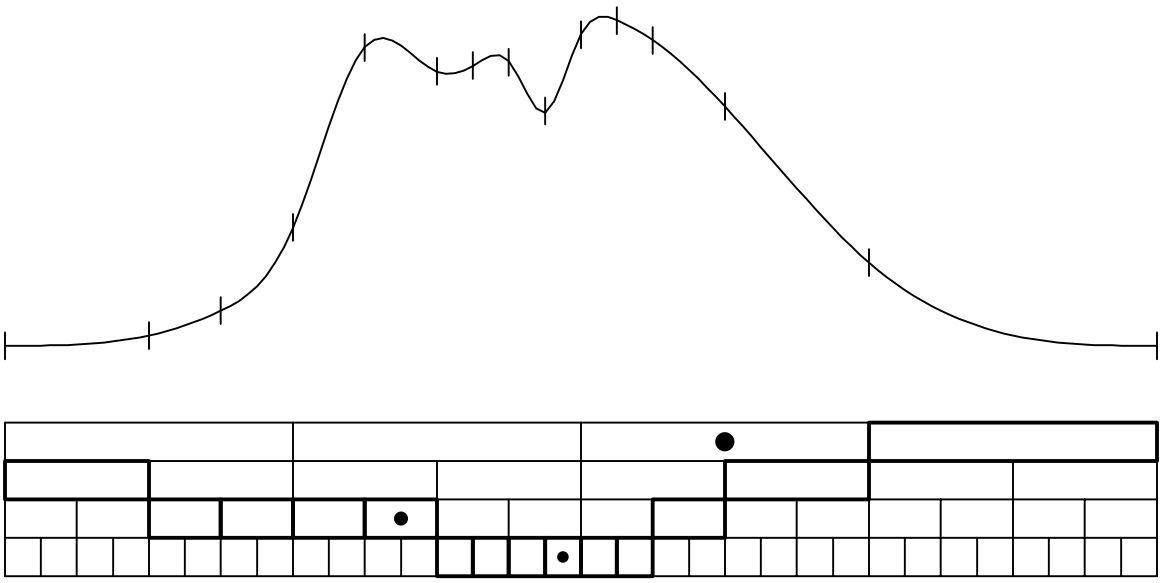


Figure 3.2.4: cubic B-spline pieces for three displacements

When there are an infinite number of nonzero displacements, evaluation depends on the restriction that the displacements die off exponentially in magnitude as the level increases—recall requirement 3.1.2, that $|D_{\ell,i}| < \mu\tau^{-\ell}$ hold for all but a finite number of displacements. The finite number of exceptions can be treated exactly as before, and what remains are *approximate* local B-spline pieces. The error in the approximation for $f(t)$ over $I_{\ell,i}$ has a magnitude smaller than

$$\epsilon(\ell) = \sum_{\ell'=\ell+1}^{\infty} \mu\tau^{-\ell'}$$

which is a convergent series for $\tau > 1$. An approximate local B-spline and its error bound are shown in Figure 3.2.5. Tighter bounds occur for the children of $I_{\ell,i}$. The children may be recursively split to whatever accuracy is desired.

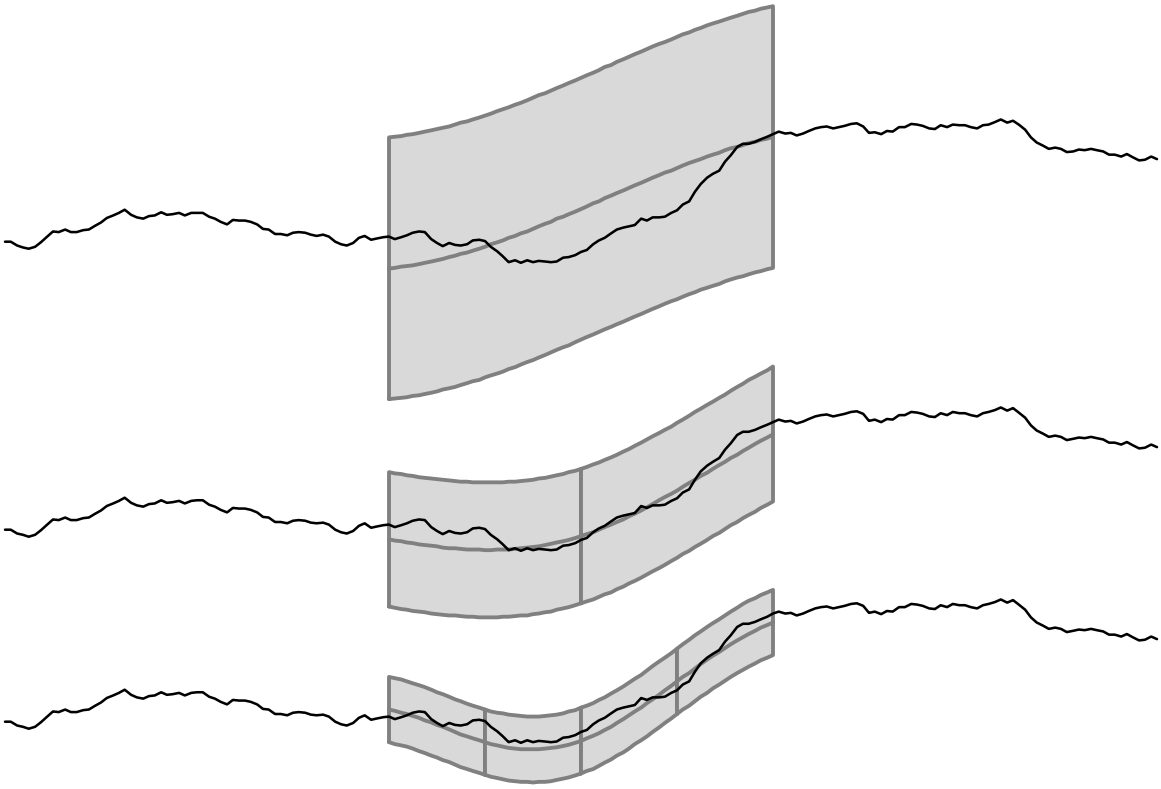


Figure 3.2.5: approximate B-spline pieces and error bounds

3.3 Interval-Query Evaluation

In order to turn dyadic splines into a useful tool, the functions must be given an interface which efficiently provides local geometric information on demand to application programs. Many applications need to access local function geometry in a way that is robust, efficient and flexible to use.

The most common way to evaluate a function is simply to compute $f(t)$ for point samples. However, this is inefficient on two counts. First, the work in computing $f(t)$ is thrown away. If a subsequent evaluation $f(t')$ is made where t' is close to t, then much of the work in computing $f(t)$ can be reused. Second, only an infinitesimal

bit of knowledge is obtained about the behavior of $f(t)$. The work in computing $f(t)$ actually reveals information on how f behaves over a neighborhood. Many applications benefit from this knowledge, and guarantees on accuracy can often be obtained only through such knowledge.

A second method to evaluate a function, called *table-lookup*, is to produce a table of range values, and use this table repeatedly when function values are needed. For example, if the domain of interest is $[0, 1]$, then a table with one hundred evenly spaced entries would be $F_i = f((i - 1/2)/100)$ with $i = 1, \ldots, 100$. This is especially useful when function computations are expensive and are used many times. For example, if each entry took 100 units of work to compute, and if each were used an average of 100 times, the effective cost per evaluation would be one unit of work plus the cost of looking up an entry in the table.

There are two problems with the use of tables. First, function use is rarely uniform and so it is inefficient to produce a uniform table. In fact, it is usually not known what the distribution of usage will be, so any fixed nonuniform table would also be inadequate. Second, there is only knowledge of the functions for the specific domain points chosen. This leads to potential inaccuracies when the table entries are used to represent the function. This can only be improved by increasing the size of the table and thus diminishing the gains in efficiency that motivated its use.

The methodology of interval queries is well suited to the task of providing an evaluation mechanism for dyadic splines. An interval query returns range positions, displacements, corner samples and range bounds for intervals in the domain hierarchy. This information is computed on demand and can be cached to avoid later recomputation. Queries generally are computed by evaluating recurrence relations that involve further interval queries.

The interval-query strategy is implemented by means of a tree of records, one per bintree interval. Each record contains the usual bookkeeping information needed to build, traverse and destroy binary trees, namely a parent and two child pointers. In addition, a record contains the position and displacement values $P_{\ell,i}$ and $D_{\ell,i}$ associated with it. These values are needed to evaluate the recurrence relations of the dyadic-spline formulation. Corner-sample and range-bound values are also stored.

The application interface to this data structure is simple and flexible. Initially, a root interval is created and returned. Subsequently, children of any interval may be queried (increasing the pool of currently-queried intervals), thus allowing any interval in the hierarchy to be reached. Edits may be performed on any queried interval, and any intervals in the query pool that are affected will receive an event. Any edit may be removed later. Finally, intervals may be released from the current-query pool.

This method has three groups of benefits:

1. flexibility of access

   - queries at diverse scales

     An application can obtain information about a dyadic-spline function for any bintree interval.

   - queries in a natural order for application

     Queries to bintree intervals can be made in almost arbitrary order. The only limitation is that a parent interval must be queried before its child.

2. robustness

   - guaranteed bounds given on range image

     Applications have the option for each interval queried to ask for guaranteed bounds on the set of range values taken on by the function over that domain interval.

   - query resolution adapts to achieve global accuracy

     An application has fine-grained control of the set of intervals queried. Using the guaranteed bounds on range over the queried intervals, computations can be made arbitrarily accurate by performing queries wherever the bounds indicate a need.

3. efficiency

   - computes only what is used

     The information required to answer interval queries is computed on demand. Information that is not needed is not computed.

   - sparse requests compute sparsely

     As a consequence of the on-demand strategy, when queries are widely spaced apart only sparse bintrees are constructed. This is similar to the kind of behavior that online computation strategies like B-spline recursive subdivision exhibit.

   - dense requests maximize reuse of computations

     Because the results of queries are cached, multiple requests to the same intervals will reuse these results. This is similar to the kind of behavior that preprocessed computation strategies like table-lookup exhibit.

   - minimum queries achieve accuracy

     In performing a computation adaptively through interval queries, an application automatically takes advantage of the on-demand computation mechanism. Thus the application has the opportunity to minimize computational work by performing queries only where the range bounds indicate a need.

Before addressing the question of effective implemenation of interval queries, one must be familiar with the notion of *interval neighborhoods*. A general interval neighborhood is defined as a union of leaf nodes for any finite tree of intervals. A simple interval neighborhood is a union of adjacent intervals at one level of the bintree. A sequence of interval neighborhoods is called a *family* if each neighborhood is a subset of its predecessor. Dependencies in the dyadic-spline recurrence formulation and other evaluations can be expressed in terms of interval neighborhoods.

Five issues must be addressed in order to make an effective implementation of interval-query evaluation:

1. evaluation of position queries

   To compute the recurrences for positions $P_{\ell,i}$ quickly, caching is essential: a naive algorithm that simply recurses without caching will experience an exponential blowup in computation cost. If a family of simple interval neighborhoods are computed in a top-down fashion, then adding caching is straightforward. A sufficiently large interval neighborhood at any level is made up of those intervals containing displacements that influence the interval containing the query interval. This is an application of Figure 3.2.2a. An example family of neighborhoods is shown in Figure 3.3.1 for a single interval query for degree $n = 2$. These neighborhoods are larger than needed near the query level, but are simple to evaluate. The unnecessary neighborhood intervals are shown as hollow dots. This minor inefficiency occurs rarely, and the excess computations are often used shortly for subsequent queries.

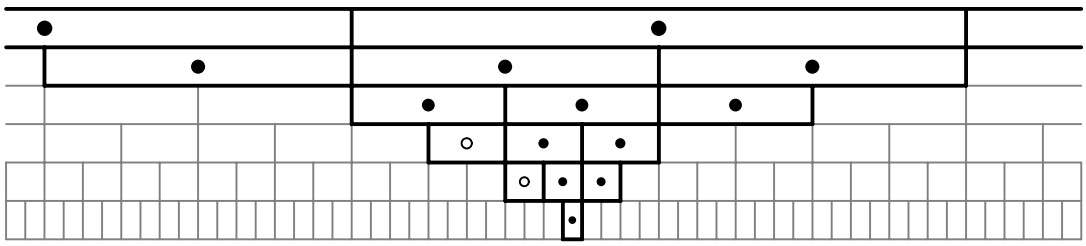


Figure 3.3.1: family of neighborhoods for $P_{\ell,i}$ query

2. evaluation of corner-sample queries

   Sample and bound computations also require that families of interval neighborhoods be maintained. These neighborhoods are used to determine the local B-spline pieces or approximations as described in section 3.2. Neighborhoods of positions and displacements are computed to levels in the tree below the one queried, and this information is recursively combined to form the requested value. For corner samples, the sample position is always kept at the center of

the neighborhood, and when no subtree in the neighborhood contains an edit, the sample is computed. A degree $n = 2$ example is shown in Figure 3.3.2. The corner sample for interval $I_{\ell,i}$ is computed, requiring neighborhoods to be generated down to the lowest influential edit (shown as a star). The positions $P_{\ell',i'}$ are computed top-down for the intervals in each neighborhood.

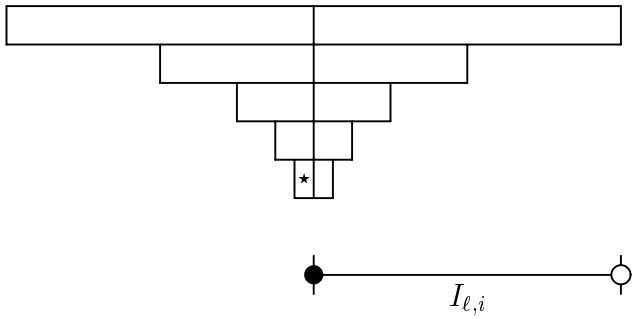


Figure 3.3.2: neighborhoods for corner query

3. evaluation of range-bound queries

   For bounds, the left and right children are both recursively traversed until no subtree in the neighborhood contains an edit, then a bound is obtained. These leaf bounds are recursively merged through unions to obtain the requested bound. Figure 3.3.3 shows the intervals whose positions $P_{\ell,i}$ are computed for two edits of a degree $n = 2$ dyadic spline. The leaves of the traversal are highlighted, and the edited intervals are marked with a star. For each leaf, the dyadic spline is locally a B-spline for which a bound was obtained and recursively merged into the total bound.

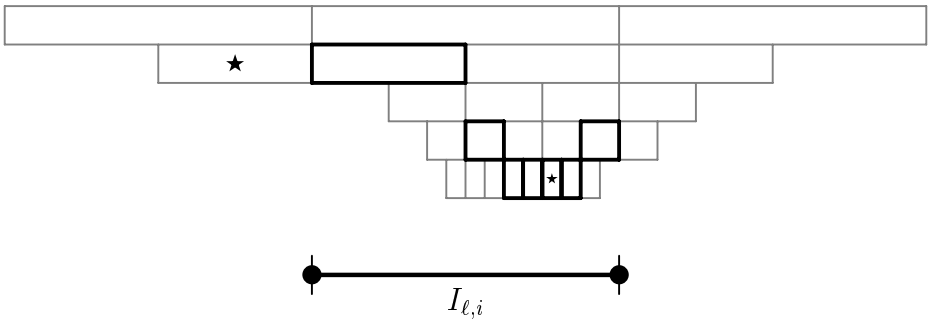


Figure 3.3.3: intervals for bound query

4. limiting memory use

   Memory use can be limited by maintaining a queue of leaf intervals that are not part of the current-query pool. These unused leaves are kept in order of latest use. Before any new interval is created, the least-recently-used intervals on the queue are destroyed until sufficient memory is available.

5. minimal updates after edits

   The final issue is to determine minimal tree updates after displacement edits. A displacement edit affects a family of neighborhoods from the root down to the level of the edit, plus the subtrees of the neighborhood at the level of the edit. For the levels at or above the edit level, only samples and bounds are affected. For levels below, position vectors $P_{\ell,i}$ are also affected. Only members of the current-query pool need be updated (along with the intervals they depend on). An example is depicted in Figure 3.3.4. The edit is marked with a star, position changes with a solid dot, and sample- or bound-only changes with a hollow dot, Note that within a few levels above the edit, the neighborhood of change typically becomes a single interval.

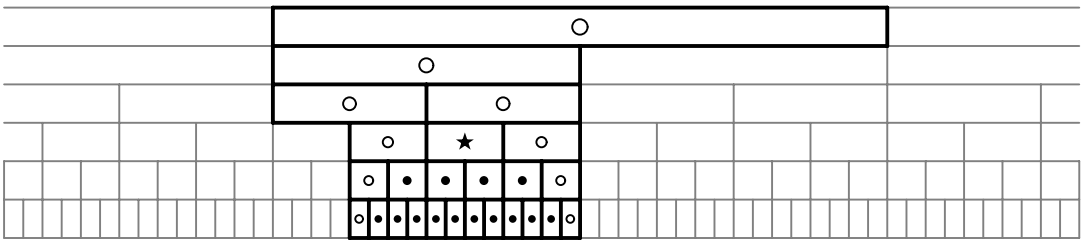


Figure 3.3.4: minimal updates after edit

Chapter 4

Function Analysis

The previous chapter detailed the process of evaluating or *synthesizing* a function when given a collection of displacements and the dyadic-spline formulation. The complementary process is the *analysis* of a given function to determine a set of displacements that will produce it. Since more than one set of displacements can produce the same function, the goal is to find a set of displacements that result in accurate approximations of the function early in the subdivision process.

This chapter first explains an approach that uses least-squares fitting to find level ℓ range positions $P_{\ell,i}$ that best predict level $\ell+1$'s range positions $P_{\ell+1,i}$. The displacements then represent the differences from the predicted values. This provides a simple bottom-up fitting process for converting uniform B-splines from a fixed, high resolution into the multiresolution displacements of a dyadic spline. After performing this fit-and-difference step, it will be seen that the displacements contain twice as much information as needed. A simple means of compacting the displacements will be introduced. This creates a multiresolution analysis framework similar to other wavelet schemes.

The process of fitting and compacting will be modified to allow variable finite filter widths that provide exact decomposition and reconstruction of the function, and will find perfect predictions when they exist. This is a significant improvement over previous linear-time smooth-wavelet schemes.

The fit-and-difference process introduced here will be used to define a *canonical form* for the displacements of a dyadic spline. It will also be shown that the fit-and-difference process can be applied directly to the displacements without computing the positions. An incremental variant on the fit-and-difference process will also be given. Collectively, these improvements to the fit-and-difference process will compliment the interval-query evaluation strategy for dyadic splines.

4.1 Least-Squares Prediction

A fundamental step in fitting dyadic splines to a given function is finding the positions P_ℓ at level ℓ that best predict the next finer positions $P_{\ell+1}$ at level $\ell+1$. The displacements $D_{\ell+1,i}$ can be set to correct the errors from the prediction. Let $\mathbf{M}$ be the degree n subdivision operator $\mathbf{M}^{(n)}$. Let the fitting operator be denoted $\mathbf{F}$. Then $P_\ell = \mathbf{F}P_{\ell+1}$ is the result of fitting. The prediction is $P'_{\ell+1} = \mathbf{M}P_\ell = \mathbf{MF}P_{\ell+1}$.

After fitting, displacements are then set to the difference from prediction

$$\begin{aligned} D_{\ell+1,i} &= P_{\ell+1} - P'_{\ell+1} \\ &= P_{\ell+1} - \mathbf{M}P_\ell \\ &= P_{\ell+1} - \mathbf{MF}P_{\ell+1} \\ &= (\mathbf{I} - \mathbf{MF})P_{\ell+1} \end{aligned}$$

This fit-and-difference step may be repeated in bottom-up order going from some lowest level $\ell_{\max}$ up to level 0. Any uniform B-spline $f(t)$ defined by $P_{\ell_{\max}}$ may be analyzed this way. The analysis produces a dyadic-spline representation that exactly reproduces the function. By performing least-squares fitting of $\mathbf{M}P_\ell$ to $P_{\ell+1}$, the displacements are minimized at each level and the displacement magnitudes are concentrated as much as possible towards the highest levels in the bintree.

To perform the least-squares fit, it is necessary to write a formula that relates one level of positions to the next. Recall operator equation 3.1.1:

$$P_{\ell+1} = \mathbf{M}P_\ell + D_{\ell+1}$$

Now least-squares fitting is formulated as follows: find the P_ℓ vector that minimizes

$$\|D_{\ell+1}\| = \|P_{\ell+1} - \mathbf{M}P_\ell\|$$

Note that $\|P_{\ell+1} - \mathbf{M}P_\ell\|$ is minimized when $(P_{\ell+1} - \mathbf{M}P_\ell) \cdot (P_{\ell+1} - \mathbf{M}P_\ell)$ is minimized. Expanding this yields

$$(P_{\ell+1} - \mathbf{M}P_\ell) \cdot (P_{\ell+1} - \mathbf{M}P_\ell) = P_\ell^T \mathbf{M}^T \mathbf{M}P_\ell - 2P_\ell^T \mathbf{M}^T P_{\ell+1} + P_{\ell+1}^T P_{\ell+1}$$

The minimum occurs when the derivative of this expression with respect to P_ℓ is zero, namely

$$2\mathbf{M}^T\mathbf{M}P_\ell - 2\mathbf{M}^T P_{\ell+1} = 0$$

This gives the optimality condition

$$\mathbf{M}^T\mathbf{M}P_\ell = \mathbf{M}^T P_{\ell+1}$$

and so the least-squares fit operator is $\mathbf{F} = (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$ giving

$$P_\ell = \mathbf{F}P_{\ell+1} = (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T P_{\ell+1}$$

Fortunately, the inverse $(\mathbf{M}^T\mathbf{M})^{-1}$ is readily computed using Fourier transforms. The operator $\mathbf{M}^T\mathbf{M}$ performs a discrete convolution with a finite, symmetric kernel $K_i = (\mathbf{M}^T\mathbf{M})_{i,0}$. The kernels for various degrees n are shown in Figure 4.1.1.

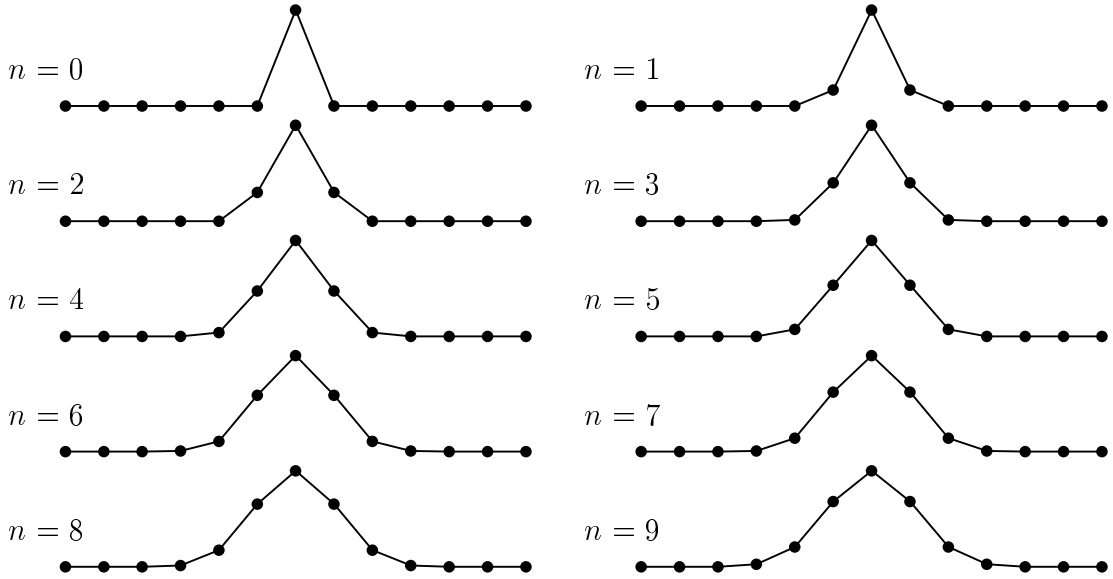


Figure 4.1.1: rows of $\mathbf{M}^T\mathbf{M}$ for various degrees

The discrete Fourier transform of K, $\kappa(x)$, is a real-valued periodic function since K is symmetric and has finite nonzero entries. Thus the transform may be written

$$\kappa(x) = \sum_i K_i cos(2\pi i x)$$

The identity convolution kernel is

$$\Delta_i = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

and has Fourier transform $\delta(x) = cos(0) = 1$. Let B be the kernel of $(\mathbf{M}^T\mathbf{M})^{-1}$ with Fourier transform $\beta(x)$. By the convolution theorem, $B * K = \Delta$ if and only if $\beta\kappa = 1$, and hence B can be found by applying the inverse Fourier transform to $1/\kappa$:

$$B_i = \int_{x\in[0,1]} \frac{cos(2\pi i x)}{\kappa(x)}$$

This integral can be computed using simple quadrature for degrees up to about $n = 15$, where the numerical errors in $B * K = \Delta$ have magnitudes less than 10^{-12}. A solution for the case $n = 9$ is depicted in Figure 4.1.2. Note that κ is very close to zero throughout the middle region, and that $1/\kappa$ forms a spike in the center. This becomes more pronounced as n increases and causes noticeable numerical degradation for $n > 15$ and complete failure at about $n = 45$. Fortunately the degrees of interest are well under $n = 15$. Note also that the magnitudes of B_i die off exponentially as i goes away from 0. Thus only a finite portion of B_i is numerically distinguishable from zero, and this is all that needs to be computed and stored.

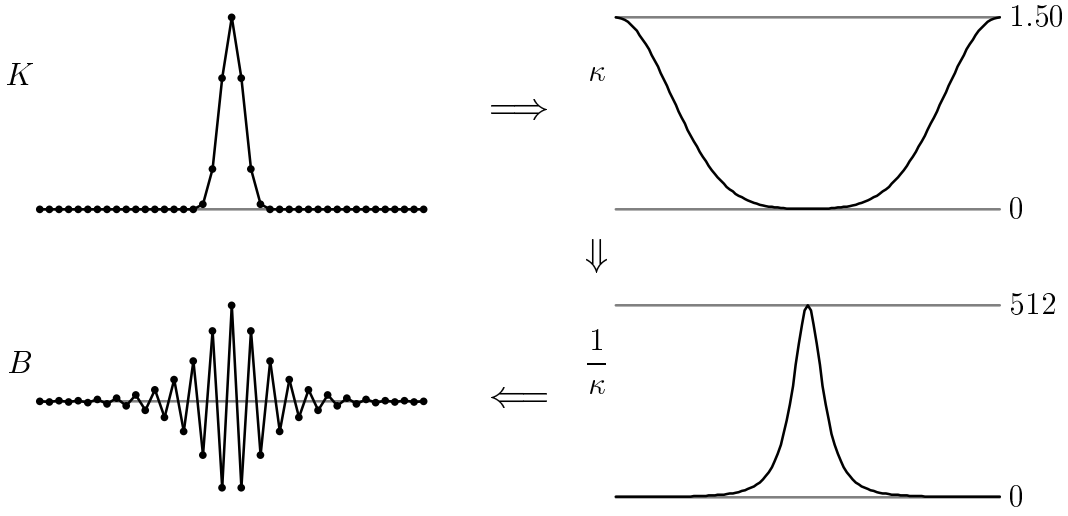


Figure 4.1.2: Fourier transform to obtain $(\mathbf{M}^T\mathbf{M})^{-1}$

After computing B_i, it is a simple matter to obtain $\mathbf{F}$. Adjacent rows of $\mathbf{F}$ differ only by a two-column translation (similar to $\mathbf{M}^T$), and so only a single row needs to be computed and stored. Figure 4.1.3 depicts one row of $\mathbf{F}$ for degrees $n = 0, \ldots, 9$.

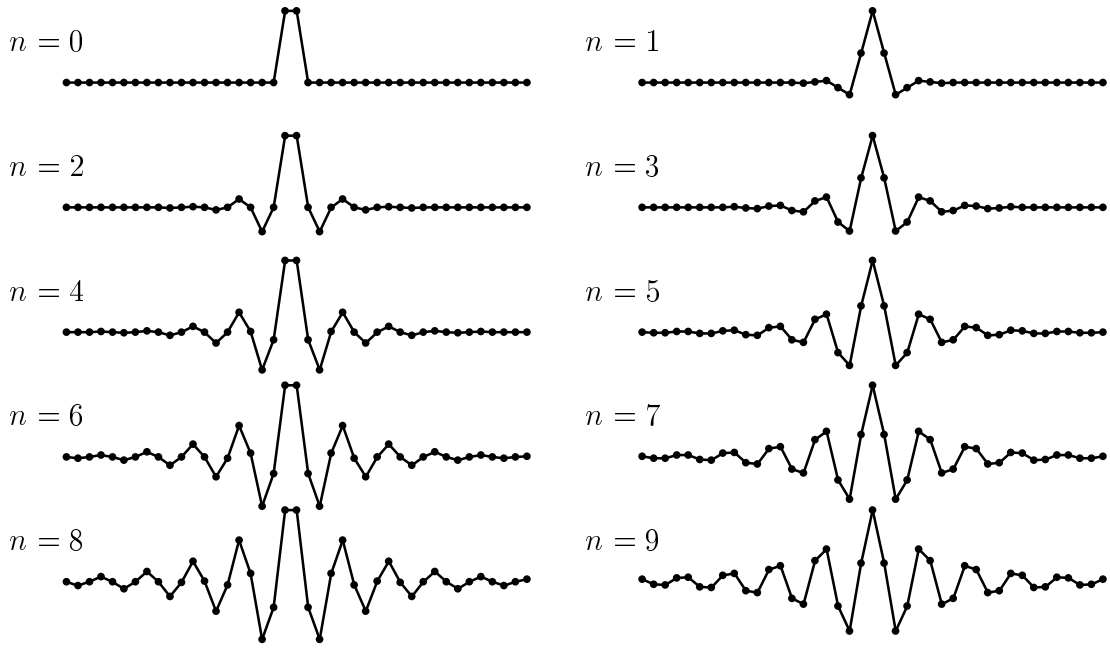


Figure 4.1.3: fit operators $\mathbf{F}$

An example of fitting and differencing is shown in Figure 4.1.4, using the cubic fit and subdivision operators. The lowest-level positions are samples of the function $f(t) = cos(2\pi t^2)/(1+t^2)$. The fit positions are shown on the left and the displacements on the right. Note that the displacements are very small at fine resolutions, indicating that the predictions are quite good in this case.

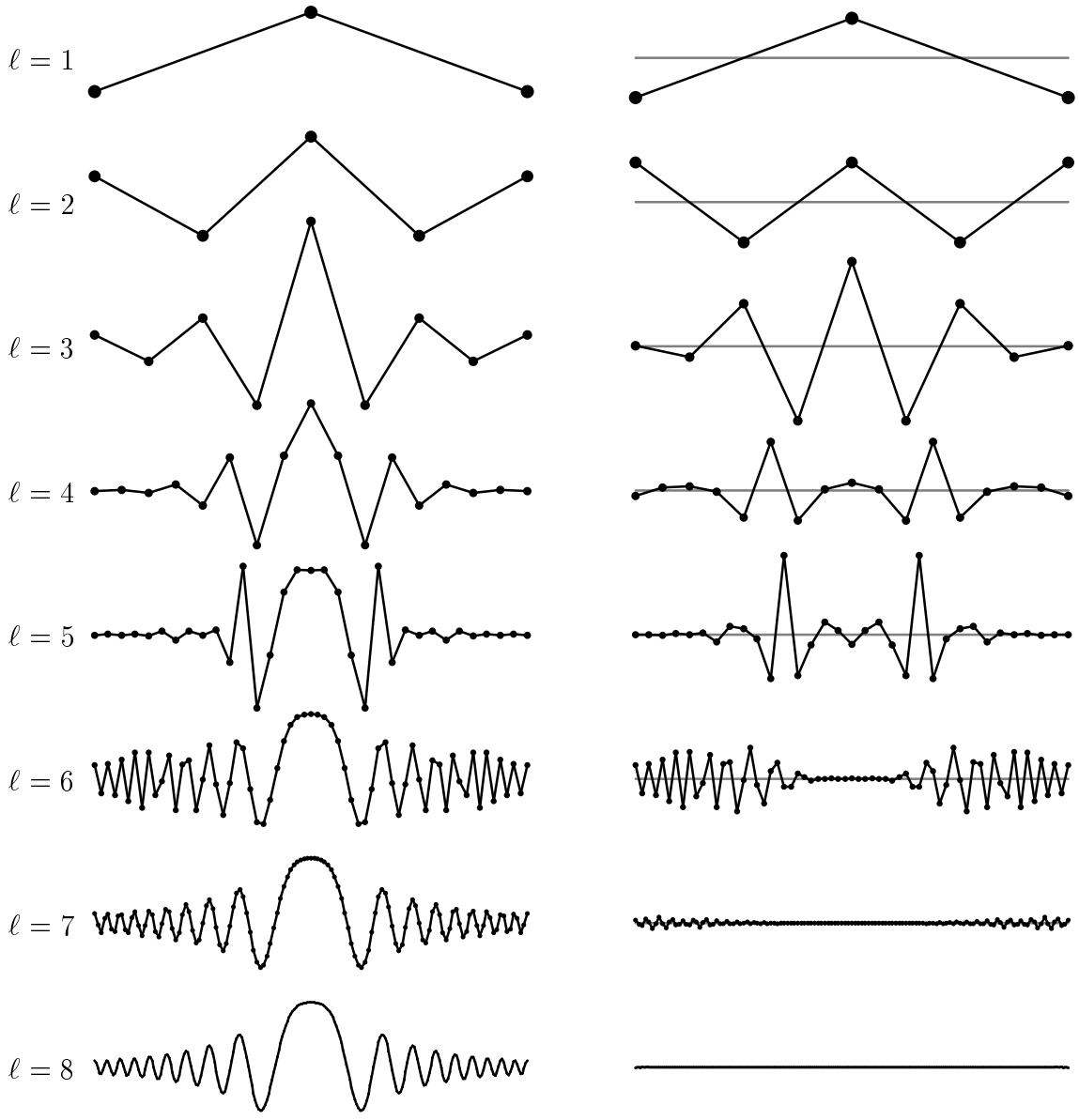


Figure 4.1.4: fitting and differencing

4.2 Compacting the Displacements

After performing a fit-and-difference step on positions $P_{\ell+1}$, the total number of values stored has increased by half. The same number of displacements $D_{\ell+1}$ exist as positions $P_{\ell+1}$, and half as many new positions P_ℓ were produced. Intuitively it appears that the displacements contain twice as many values as one might expect. This is in fact the case, as will be seen in this section.

If the differences from the fit predictions, $D_{\ell+1} = (\mathbf{I} - \mathbf{MF})P_{\ell+1}$, are to be compacted to half as many values, then a compaction operator $\mathbf{C}$ and its complimentary expansion operator $\mathbf{E}$ are needed. These operators should satisfy the following

$$\begin{bmatrix} \mathbf{F} \\ \mathbf{C}(\mathbf{I} - \mathbf{MF}) \end{bmatrix} [\mathbf{M}|\mathbf{E}] = \begin{bmatrix} \mathbf{I} & 0 \\ 0 & \mathbf{I} \end{bmatrix}$$

In other words, the operation of fitting and compacting the differences from prediction should be the inverse of the operation of subdividing and expanding the compacted differences. The shorthand formula above expands into four requirements:

$$\begin{aligned} \mathbf{FM} &= \mathbf{I} \\ \mathbf{FE} &= 0 \\ \mathbf{C}(\mathbf{I} - \mathbf{MF})\mathbf{M} &= 0 \\ \mathbf{C}(\mathbf{I} - \mathbf{MF})\mathbf{E} &= \mathbf{I} \end{aligned}$$

We already know that

$$\begin{aligned} \mathbf{FM} &= (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{M} \\ &= \mathbf{I} \end{aligned}$$

The first requirement on $\mathbf{C}$ is automatically satisfied:

$$\begin{aligned} \mathbf{C}(\mathbf{I} - \mathbf{MF})\mathbf{M} &= \mathbf{C}(\mathbf{M} - \mathbf{MFM}) \\ &= \mathbf{C}(\mathbf{M} - \mathbf{MI}) \\ &= 0 \end{aligned}$$

If $\mathbf{E}$ has been chosen so that $\mathbf{FE} = 0$, then the second requirement on $\mathbf{C}$ simplifies:

$$\begin{aligned} \mathbf{C}(\mathbf{I} - \mathbf{MF})\mathbf{E} &= \mathbf{I} \\ \mathbf{C}(\mathbf{E} - \mathbf{MFE}) &= \mathbf{I} \\ \mathbf{C}(\mathbf{E} - \mathbf{M}0) &= \mathbf{I} \\ \mathbf{CE} &= \mathbf{I} \end{aligned}$$

So the requirements that remain are

$$\begin{aligned} \mathbf{FE} &= 0 \\ \mathbf{CE} &= \mathbf{I} \end{aligned}$$

A simple method suffices to ensure $\mathbf{FE} = 0$ and $\mathbf{CE} = \mathbf{I}$. The idea is to make $\mathbf{E}$ equal to the transpose of $\mathbf{F}$ and then negate every even row, and to make $\mathbf{C}$ equal to the transpose of $\mathbf{M}$ and then negate every even column. This works as stated for even degree n operators $\mathbf{M}$ and $\mathbf{F}$. For odd degrees the columns of $\mathbf{E}$ and the rows of $\mathbf{C}$ must be shifted forward one row and column, respectively. Formally, the method is

$$\mathbf{C}_{i,j} = \begin{cases} -\mathbf{M}_{j,i} & \text{if } j \text{ even} \\ \mathbf{M}_{j,i} & \text{if } j \text{ odd} \end{cases}$$

$$\mathbf{E}_{i,j} = \begin{cases} -\mathbf{F}_{j,i} & \text{if } i \text{ even} \\ \mathbf{F}_{j,i} & \text{if } i \text{ odd} \end{cases}$$

for n even and

$$\mathbf{C}_{i,j} = \begin{cases} -\mathbf{M}_{j,i-1} & \text{if } j \text{ even} \\ \mathbf{M}_{j,i-1} & \text{if } j \text{ odd} \end{cases}$$

$$\mathbf{E}_{i,j} = \begin{cases} -\mathbf{F}_{j-1,i} & \text{if } i \text{ even} \\ \mathbf{F}_{j-1,i} & \text{if } i \text{ odd} \end{cases}$$

for n odd.

The verification that this works is straightforward. Recall that $\mathbf{F}$ really has just a single row, and that the operator is made up of copies of that row that are shifted by multiples of two places. Now, notice that by multiplying $\mathbf{F}$ against its transpose $\mathbf{F}^T$, the one replicated row of $\mathbf{F}$ is multiplied against copies of itself that are shifted by multiples of two places. Because the row is symmetric, when every other entry of the shifted copies is negated the product is zero. In the case of odd n, the copies must be shifted one place in order for this to work. Thus $\mathbf{E}$ has been constructed so that $\mathbf{FE} = 0$. A picture of this cancellation is shown in Figure 4.2.1.

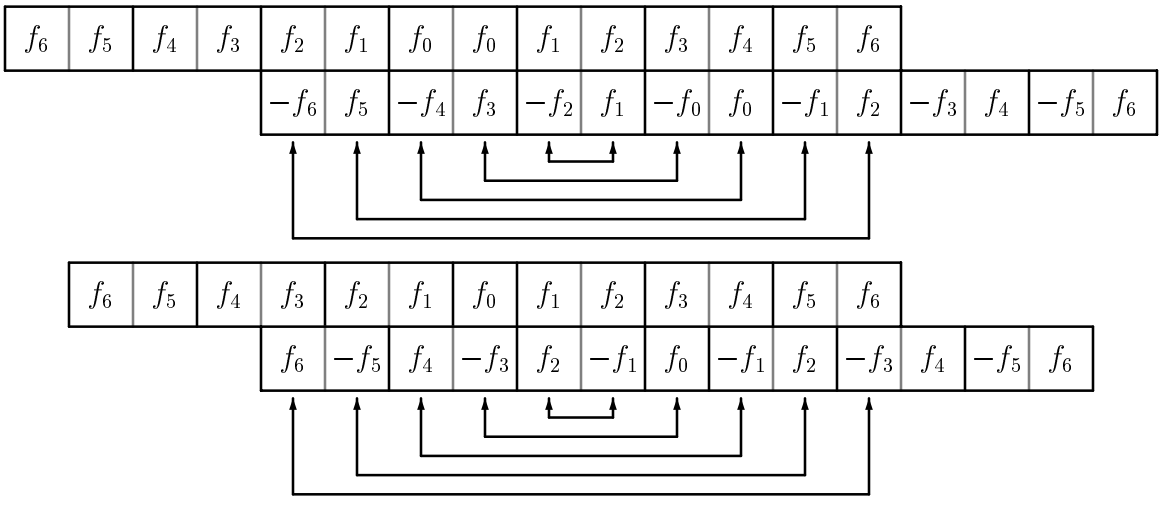


Figure 4.2.1: cancellation of $\mathbf{FE}$: (a) n even (b) n odd

Verification of $\mathbf{CE} = \mathbf{I}$ is also straightforward. The columns of $\mathbf{C}$ are negated in exact correspondence with the rows of $\mathbf{E}$, and thus the negations cancel and give $\mathbf{CE} = \mathbf{M}^T\mathbf{F}^T = (\mathbf{FM})^T = \mathbf{I}$. For odd n the extra shift results in a shift in $\mathbf{I}$, which is still $\mathbf{I}$.

Note also that $\mathbf{CM} = 0$. This follows using the same argument that demonstrated $\mathbf{FE} = 0$. This means that the operation of compacting the differences from prediction is the same as compacting the positions:

$$\begin{aligned} \mathbf{C}(\mathbf{I} - \mathbf{MF}) &= \mathbf{CI} - \mathbf{CMF} \\ &= \mathbf{C} - 0\mathbf{F} \\ &= \mathbf{C} \end{aligned}$$

The operators $\mathbf{F}$, $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{E}$ now form a multiresolution analysis filter bank as described in section 2.4. The scaling functions $\phi_{\ell,i}(t)$ are just the uniform B-spline basis functions. The wavelet functions $\psi_{\ell,i}(t)$ can be constructed using the filter bank. Scaling and wavelet functions are shown in Figure 4.2.2 for degrees $n = 0, \ldots, 9$.

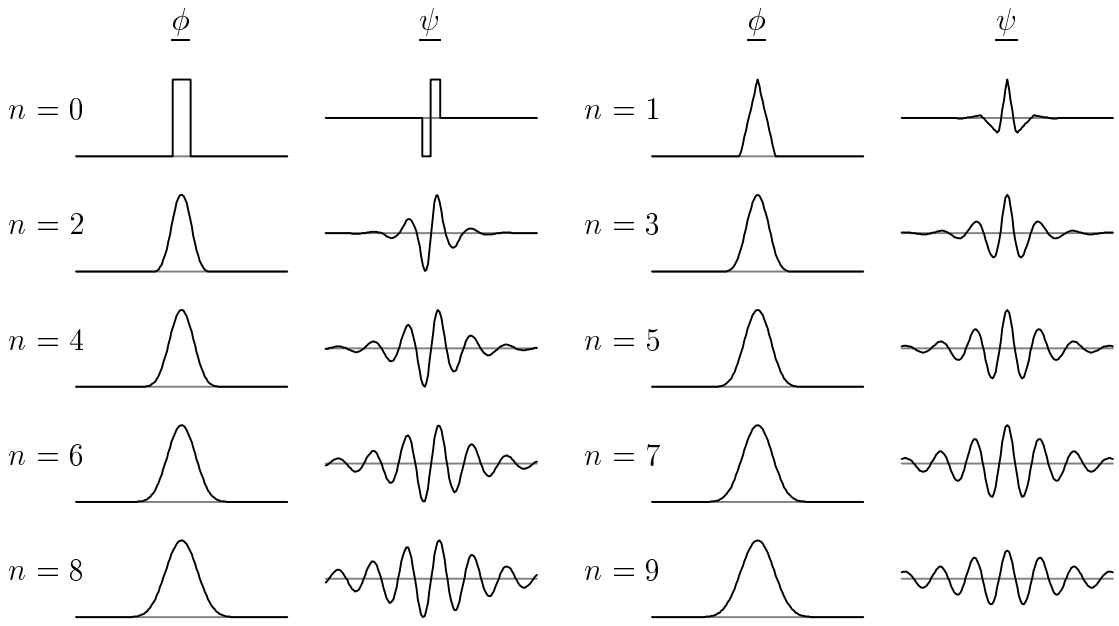


Figure 4.2.2: ϕ and ψ for $n = 0, \ldots, 9$

4.3 Finite-Width Filters

Although the magnitudes of the fit operator entries $\mathbf{F}_{i,j}$ die off exponentially away from $j = 2i$, a large number of these entries must come into play in order

to get a numerically accurate reconstruction. A better approach is to construct a small, finite number of nonzero row entries in $\mathbf{F}$ that provide exact reconstruction, find perfect predictions when possible, and are close to the least-squares fit operator $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$.

Any operator $\mathbf{F}$ that satisfies $\mathbf{FM} = \mathbf{I}$ will find perfect predictions whenever possible. If P_ℓ perfectly predicts $P_{\ell+1}$, then $P_{\ell+1} = \mathbf{M}P_\ell$. The fit of $P_{\ell+1}$ is then

$$\begin{aligned} \mathbf{F}P_{\ell+1} &= \mathbf{FM}P_\ell \\ &= \mathbf{I}P_\ell \\ &= P_\ell \end{aligned}$$

For the construction of the $\mathbf{E}$ operator in the multiresolution analysis filter bank, any fit operator $\mathbf{F}$ will work as long as it is made up of a single, symmetric row that is copied by shifting a multiple of two places, and if in addition $\mathbf{FM} = \mathbf{I}$ holds. Recall from section 4.2 that filters $\mathbf{F}$, $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{E}$ must satisfy

$$\begin{aligned} \mathbf{FM} &= \mathbf{I} \\ \mathbf{FE} &= 0 \\ \mathbf{CE} &= \mathbf{I} \end{aligned}$$

The first requirement is met by assumption. The second is met because $\mathbf{F}$ is made up of a single, symmetric row that is copied by shifting a multiple of two places. The third assumption is met automatically by the construction of $\mathbf{C}$ and $\mathbf{E}$ when $\mathbf{FM} = \mathbf{I}$.

Hence we would like to find a finite, symmetric set of row entries to $\mathbf{F}$ that satisfies $\mathbf{FM} = \mathbf{I}$ and is close to $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$. The approach taken here is to start with a finite portion of row zero of $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$, append a small number of additional entries to this finite row, and form a square matrix of constraints on these additional entries.

Before giving the general case, consider the situation when $n = 2$ and four entries of $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$ are used. Row zero of $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$ is

$$\left[\ldots, 0, \tfrac{2}{3}, \tfrac{2}{3}, 0, \ldots\right]$$

Appending two additional entries, a and b, to both sides of the four central entries gives the eight-entry row

$$\left[a, b, 0, \tfrac{2}{3}, \tfrac{2}{3}, 0, b, a\right]$$

Now a square system of constraints can be extracted from $\mathbf{FM} = \mathbf{I}$:

$$\Big[a, b, 0, 2/3, \underline{2/3}, 0, b, a\Big] \begin{bmatrix} & & \vdots & & \\ & 1 & 0 & 0 & \\ & 3 & 0 & 0 & \\ & 3 & 1 & 0 & \\ & 1 & 3 & 0 & \\ \cdots & 0 & \underline{3} & 1 & \cdots \\ & 0 & 1 & 3 & \\ & 0 & 0 & 3 & \\ & 0 & 0 & 1 & \\ & & \vdots & & \end{bmatrix} = 4I$$

So

$$\begin{bmatrix} 1 & 3 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} -\frac{2}{3} \\ 0 \end{bmatrix}$$

The solution has $a = 1/12$ and $b = -1/4$:

$$\Big[\tfrac{1}{12}, -\tfrac{1}{4}, 0, \tfrac{2}{3}, \tfrac{2}{3}, 0, -\tfrac{1}{4}, \tfrac{1}{12}\Big]$$

In general, a square system will be formed when the total number of nonzero entries in the row of $\mathbf{F}$ is

$$m = n + 2 - 2(n \bmod 2)$$

for some $k \geq 0$, and when the number of extra entries is $\min\{n, m\}$. Any remaining central entries are copied from row zero of $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$. A square system is determined by $\mathbf{FM} = \mathbf{I}$. Some of the resulting fit operator rows are shown in Figure 4.3.1.

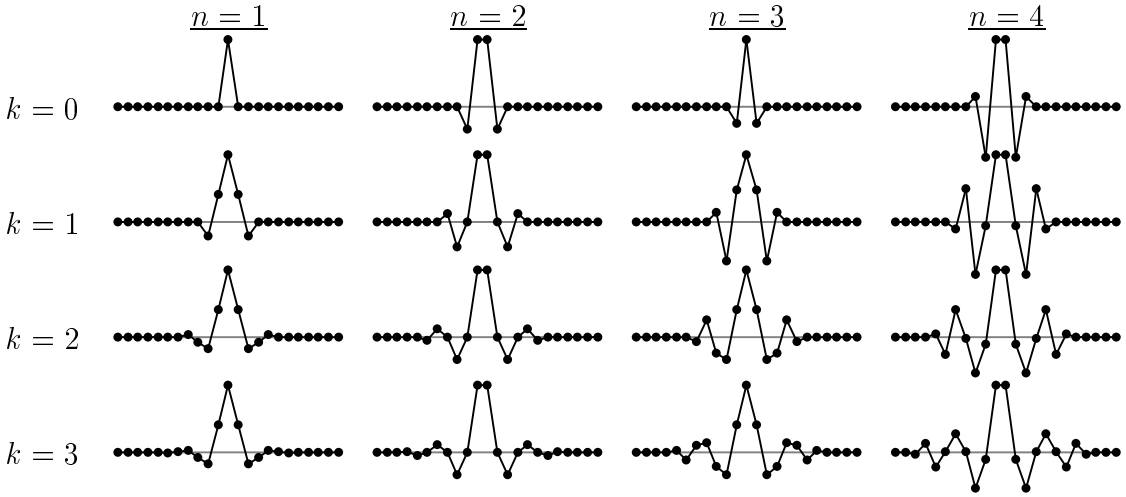


Figure 4.3.1: finite fit operators

The only loss in going to finite versions of the fit operators is that they no longer produce the least-squares fit. As the width of the fit operator increases, the fit comes closer to the least-squares fit. The residual displacements and energy are shown in Figure 4.3.2 for various quadratic filter widths applied to a random signal. Note that the energy is nearly minimized for small filter widths.

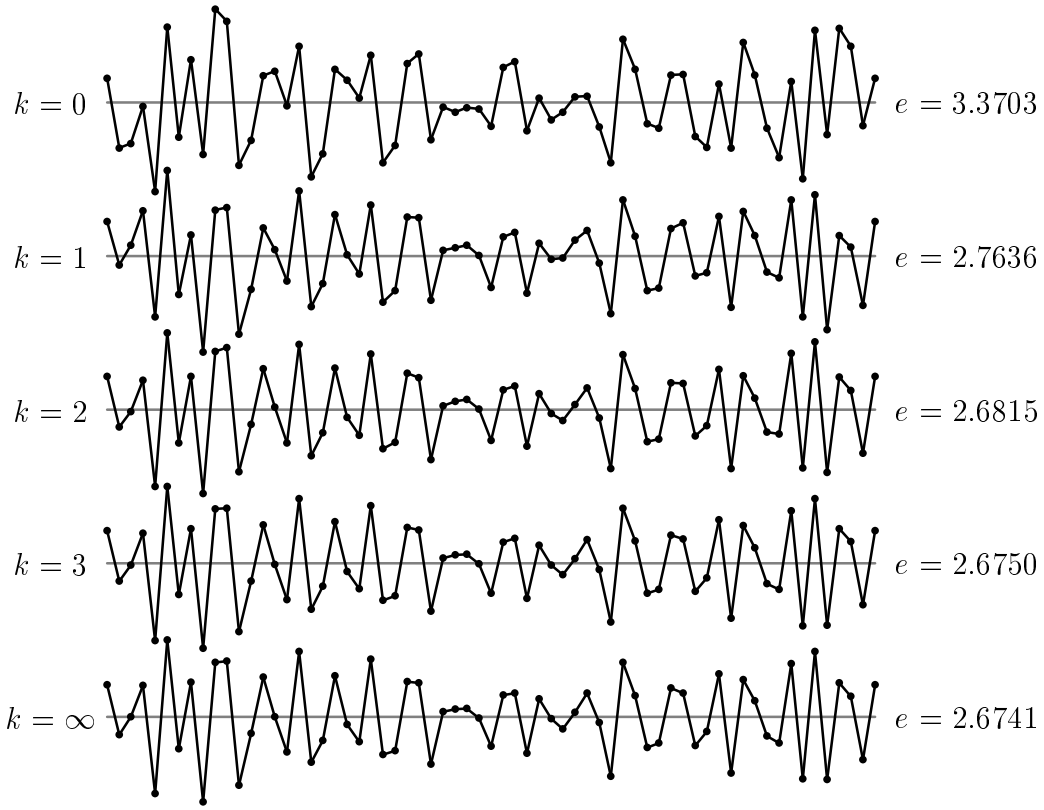


Figure 4.3.2: residual displacement energy versus filter width

4.4 Improving the Displacements

Several improvements can be made to the storage and processing of displacements based on the fit-and-difference process:

- displacements can be converted to a canonical form
- fit-and-difference processing can be applied directly to displacements instead of to positions
- fitting can be done incrementally

These improvements have significant benefits for interval-query evaluation. The canonical form ensures that multiresolution queries return good local approximations at all scales and localities. Direct application of fit-and-difference processing to displacements improves the efficiency of conversion to the canonical form. Finally, incremental fitting allows the work of conversions to be performed precisely for those displacements that change during interactive use or during the top-down approximation processing described in section 5.2.

4.4.1 Canonical Displacements

One complication with dyadic splines is that more than one configuration of displacements can represent the same function. Consider the displacements shown as arrows in Figure 4.4.1 under quadratic subdivision. On the left, the single nonzero displacement is $D_{0,0} = 1$. On the right, the only nonzero displacements are $\hat{D}_{1,-1} = 1/4$, $\hat{D}_{1,0} = 3/4$, $\hat{D}_{1,1} = 3/4$, and $\hat{D}_{1,2} = 1/4$. These two displacement arrangements produce the same positions $P_{1,i} = \hat{P}_{1,i}$ and the same limit function.

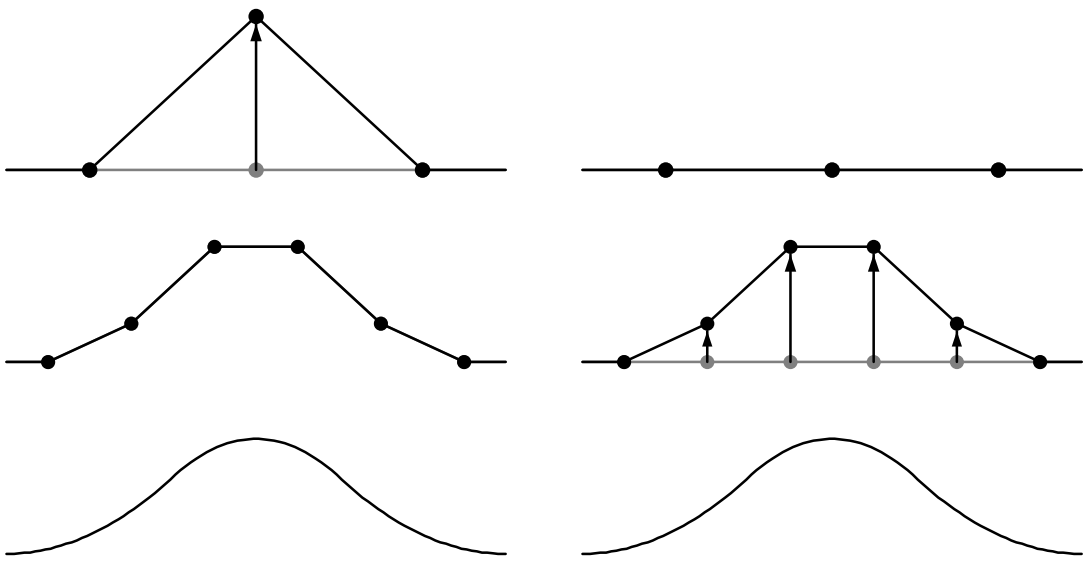


Figure 4.4.1: different displacements for the same function

In this example, it is clear that the level $\ell = 0$ displacements on the left give more accurate information about the function. In fact, they perfectly predict the positions at subsequent levels, since all lower displacements are zero. In general, displacements should give perfect predictions when possible, and should otherwise minimize the errors from the predicted positions. Furthermore, it is best to have a unique, optimal set of displacements for any limit function.

The bottom-up conversion process is suitable for producing unique, optimal displacements. One algorithm to put a set of displacements into a canonical, optimized form is simply to produce a fine set of positions $P_{\ell_{\max}}$ by applying the dyadic-spline formulation to the original displacements, and then apply the bottom-up fit-and-difference processing to get the canonical displacements.

This kind of conversion facilitates interactive design because overall trends that accumulate at the detail level will propagate to coarser levels automatically. This allows powerful control mechanisms such as sculpting to be used efficiently. Compression is a natural byproduct of this since energy tends to concentrate at high levels in the bintree, and significant local energy tends to be sparse in lower parts of the tree.

4.4.2 Displacement Fitting

Recall the fit-and-difference step from section 4.1:

$$\begin{aligned} P_\ell &= \mathbf{F}P_{\ell+1} \\ D_{\ell+1} &= (\mathbf{I}-\mathbf{MF})P_{\ell+1} \end{aligned}$$

This defines the optimized positions and displacements in bottom-up order. The unoptimized positions $\overline{P}_\ell$ and unoptimized displacements $\overline{D}_{\ell+1}$ can be written as

$$P_{\ell+1} = \mathbf{M}\overline{P}_\ell + \overline{D}_{\ell+1}$$

Now the optimized displacements may be rewritten as

$$\begin{aligned} D_{\ell+1} &= (\mathbf{I}-\mathbf{MF})P_{\ell+1} \\ &= (\mathbf{I}-\mathbf{MF})(\mathbf{M}\overline{P}_\ell + \overline{D}_{\ell+1}) \\ &= (\mathbf{M}-\mathbf{MFM})\overline{P}_\ell + (\mathbf{I}-\mathbf{MF})\overline{D}_{\ell+1} \\ &= (\mathbf{M}-\mathbf{MI})\overline{P}_\ell + (\mathbf{I}-\mathbf{MF})\overline{D}_{\ell+1} \\ &= (\mathbf{I}-\mathbf{MF})\overline{D}_{\ell+1} \end{aligned}$$

The fit of the unoptimized displacements must be added to the unoptimized displacements one level up in order to maintain the same final positions:

$$\hat{D}_\ell = \overline{D}_\ell + \mathbf{F}\overline{D}_{\ell+1}$$

The new positions at level ℓ that correspond to $\hat{D}_\ell$ are

$$\hat{P}_\ell = \overline{P}_\ell + \mathbf{F}\overline{D}_{\ell+1}$$

To verify that these new positions are in fact optimal, note that

$$\begin{aligned} \hat{P}_\ell &= \overline{P}_\ell + \mathbf{F}\overline{D}_{\ell+1} \\ &= \mathbf{FM}\overline{P}_\ell + \mathbf{F}\overline{D}_{\ell+1} \\ &= \mathbf{F}(\mathbf{M}\overline{P}_\ell + \overline{D}_{\ell+1}) \\ &= \mathbf{F}P_{\ell+1} \\ &= P_\ell \end{aligned}$$

Thus the fit-and-difference processing can be applied directly to the displacements without computing positions $P_{\ell+1}$. In effect, this moves as much displacement energy as possible from level $\ell+1$ to level ℓ.

4.4.3 Incremental Fitting

So far, the fit-and-difference processing has been described as a global operation on the dyadic-spline displacements. This is not an adequate solution for interactive applications such as editing, painting and scupting, nor is it suitable for sparse top-down approximation. Two further properties of the dyadic-spline formulation must be exploited. These properties are the locality of changes during editing, and the typical sparseness of nonzero displacements.

To exploit locality, it is necessary to use a finite-width fit operator $\mathbf{F}$. In the previous section the global fit-and-difference step was applied directly to the displacements:

$$\begin{aligned} D_{\ell+1} &= (\mathbf{I}-\mathbf{MF})\overline{D}_{\ell+1} \\ \hat{D}_\ell &= \overline{D}_\ell + \mathbf{F}\overline{D}_{\ell+1} \end{aligned}$$

This produces a whole level of optimized displacements $D_{\ell+1}$, and adds the fit energy to the next higher level of displacements. Now suppose exactly one of these optimized displacements is modified:

$$\overline{D}'_{\ell+1,i} = D_{\ell+1,i} + \begin{cases} d & \text{if } i = i_0 \\ 0 & \text{otherwise} \end{cases}$$

Optimizing again gives

$$\begin{aligned} D'_{\ell+1} &= (\mathbf{I}-\mathbf{MF})\overline{D}'_{\ell+1} \\ &= D_{\ell+1} + d(\mathbf{I}-\mathbf{MF})_{\cdot,i_0} \end{aligned}$$

and

$$\begin{aligned} \hat{D}'_\ell &= \hat{D}_\ell + \mathbf{F}\overline{D}'_{\ell+1} \\ &= \hat{D}_\ell + d\mathbf{F}_{\cdot,i_0} \end{aligned}$$

(where the notation $\mathbf{A}_{\cdot,i}$ refers to extracting the ith column of operator $\mathbf{A}$).

The effect of a single displacement edit is shown in Figure 4.4.2 for width 8 quadratic fitting. The gray displacements depict the state right after a single edit has been made. These are the unoptimized displacements denoted $\overline{D}'_{\ell+1}$ and $\hat{D}_\ell$ above. After reoptimizing at level $\ell+1$, $D'_{\ell+1}$ and $\hat{D}'_\ell$ are produced, as shown in black. The level ℓ displacement changes must also be reoptimized, causing changes to level $\ell-1$, $\ell-2$, etc. Thus a bottom-up fit-and-difference process is needed, but only a neighborhood of changes need be considered at each level. Fortunately, as these changes propagate upward, the affected neighborhoods do not grow indefinitely in size. This

reasoning is similar to that used to determine the neighborhoods that influence a given output position during synthesis.

For the sake of efficiency, the reoptimization of a given level can be delayed until it is needed. For example, if the unoptimized displacements on a given level ℓ are updated only after $2^{\ell_{\max}-\ell}$ edits, the total update time will be a small constant times the number of edits. This requires that a list be kept of the displacement neighborhoods that need to be reoptimized.

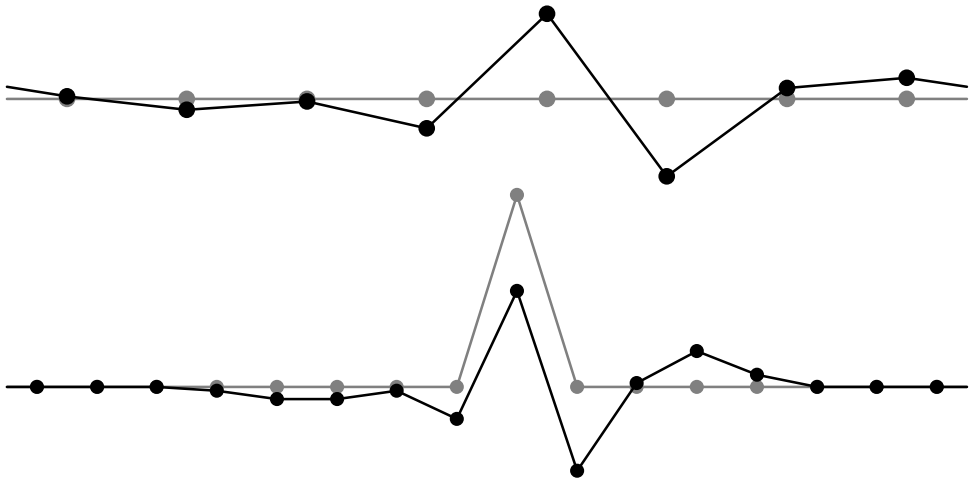


Figure 4.4.2: fit update for one edit
(gray) edit (black) reoptimized

Chapter 5

Function Approximation

The last chapter presented a technique for performing multiresolution analysis. The transform uses decomposition and reconstruction filter banks to convert uniform B-splines of high resolution into the multiresolution displacements of dyadic splines and back again, while optimizing level-to-level predictions and removing the displacement redundancy. This whole process is exact, and no information is lost. This chapter generalizes that fitting process to allow approximate conversion of an input function into the dyadic-spline representation. This will provide a tool to trade off the level of complexity in the dyadic spline against the accuracy of the approximation.

There are many ways to measure complexity and accuracy of an approximation, each driven by the applications that make use of the tradeoff between the two. Complexity can be measured by the number of nonzero compacted displacements, the amount of work or memory use during computations, or the number of bits required for transmission or storage between computations. Accuracy is generally measured by a norm on the difference between the actual and approximate function. The three norms most commonly used are the L_1 (mean-magnitude) norm, the L_2 (root-mean-square) norm and the L_∞ (supremum) norm. For sampled data (such as scanned images), the discrete versions of these norms are used.

Ideally, the complexity should be minimized for a given accuracy. Conversely, the accuracy should be maximized for a given complexity. A more global view is that the graph of accuracy versus complexity should be optimized over a range of interest. For example, a plot of signal-to-noise ratio versus number of bits is commonly used to compare lossy image compression methods.

This chapter presents two generic methods of trading off the number of nonzero compacted displacements against a norm on the approximation error. The first method works bottom-up, starting with a high-resolution set of positions. The positions are converted to dyadic-spline compacted displacements using the fit-and-compact filter steps. The complexity is then reduced by zeroing the compacted displacements that have the least impact on the error. The second method works top-down, using estimates on the target function over intervals. Compacted displacements are added to neighborhoods where errors are the greatest.

The approximation techniques have numerous applications. For shape design, interactive editing such as sculpting benefits since the complexity normally grows to intractable levels during a design session. Approximation can be applied as editing

takes place to maintain an upper limit on this complexity. For interactive display, complexity must also be strictly limited in order to maintain high frame rates. A simplistic lossy bit compression method can also be derived from these approximation techniques.

5.1 Bottom-Up Approximation

For bottom-up approximation, it is assumed that positions $P_{\ell_{\max}}$ have been obtained as samples or by fitting to a continuous target function $g(t)$ over domain $t \in [0,1]$ as described earlier in section 2.2.5. Let the dyadic-spline displacements $D_{\ell,i}$ be in canonical form for some degree n and filter width parameter k, and let $Q_{\ell,i}$ be the compacted form of these displacements. For simplicity, assume that the dyadic spline $f(t)$ obtained from these displacements is periodic over $[0,1]$, and therefore for each level ℓ only a finite number of indices, $i = 0, \ldots, 2^\ell - 1$, need be considered. The approximation algorithm will set some of the compacted displacements to zero until a desired complexity or error threshold is crossed.

Clearly it is impractical to search through all subsets of compacted displacements to find an optimum subset. A less expensive approach is to use a greedy algorithm that zeros out the compacted displacement that leaves the error the smallest. Even this is expensive, since all nonzero displacements might need to be examined at each step to determine their impact on the error—and these impacts can change at every step. Increasing the efficiency of this type of greedy algorithm is highly dependent on the type of norm used. An *expected-error* greedy algorithm performs the greedy step based on error measured with respect to the current approximation. This improves efficiency since the expected error from zeroing a compacted displacement remains constant throughout the approximation processing. The expected-error approach works efficiently with any norm.

The expected-error greedy algorithm is described as follows. For each bintree level $\ell = 0, \ldots, \ell_{\max}$, compute the expected error ϵ_ℓ caused by changing a compacted displacement from one to zero

$$\epsilon_\ell = \|\psi_{\ell,0}\|$$

where $\psi_{\ell,0}(t)$ is the wavelet corresponding to compacted displacement $Q_{\ell,0}$. The norm of $\psi_{\ell,0}(t)$ is identical to all the other wavelets $\psi_{\ell,i}(t)$ at level ℓ (the definition and computation of these wavelets was described in sections 4.2 and 4.3). The expected error $E_{\ell,i}$ from zeroing a compacted displacement $Q_{\ell,i}$ is then

$$E_{\ell,i} = |Q_{\ell,i}|\epsilon_\ell$$

Now a sequence of compacted displacements $Q_{\ell_1,i_1}, Q_{\ell_2,i_2}, \ldots$ are zeroed until N nonzero compacted displacements remain or until the total expected error $\sum E_{\ell_j,i_j}$ reaches a threshold $E_{\max}$. The values N and $E_{\max}$ are parameters to the algorithm.

The ideal order of the sequence (ℓ_j, i_j) is obtained by sorting the set of expected errors, $\left\{E_{\ell,i} \mid \ell = 0, \ldots, \ell_{\max}; i = 0, \ldots, 2^\ell - 1\right\}$, from smallest to largest. It is reasonable to approximate this order by bucket sorting, so that the total running time is proportional to the number of initial compacted displacements.

Some results of the expected-error greedy algorithm are depicted on the next page, in Figure 5.1.3. The original function f, shown at top, is the central row of pixels of the well-known mandrill image, shown in Figure 5.1.2. The functions below that are approximations $\hat{f}$ produced by the algorithm using the L_2 norm with degree $n = 2$ and filter width $k = 1$. To the right of each approximation there is a plot of the the error $\hat{f} - f$, the norm of this error $e = \left\|\hat{f} - f\right\|_2$, and number of nonzero compacted displacements N.

A plot of accuracy versus number of compacted displacements is shown below in Figure 5.1.1. Accuracy is measured as the mean-square signal-to-noise ratio in decibels

$$10 \log_{10}(\|f - f_a\|_2^2 / \left\|\hat{f} - f\right\|_2^2)$$

where f_a is the average value f takes on. The black curve is for the expected-error greedy algorithm. For comparison, a plot for the true greedy algorithm is shown in gray.

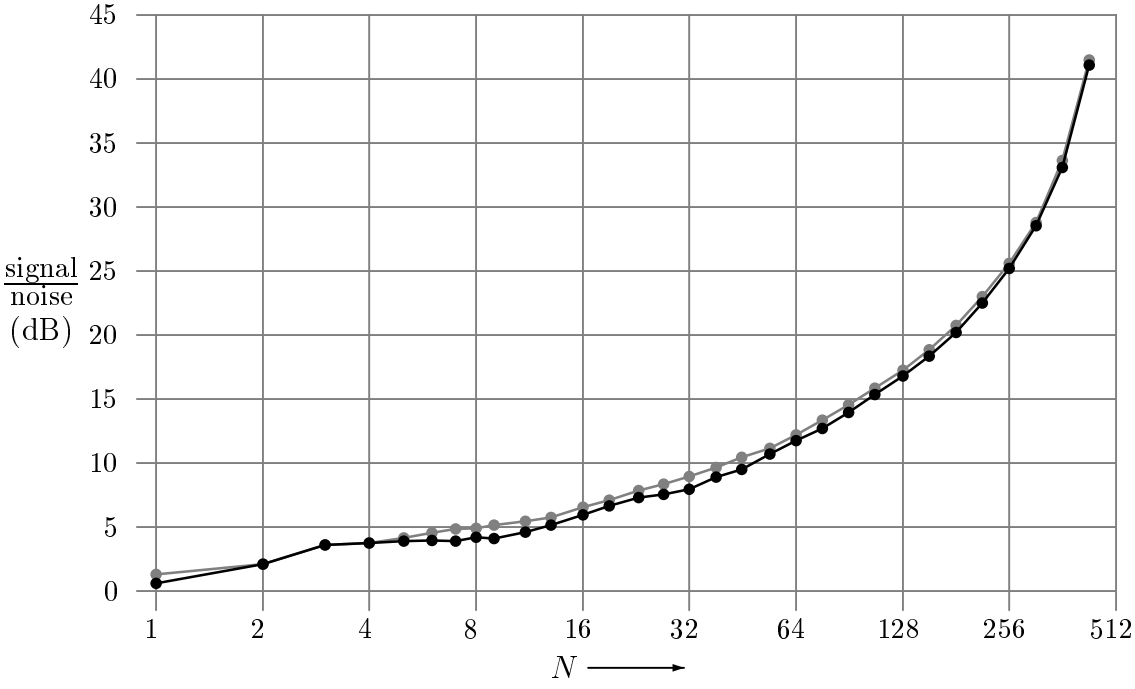


Figure 5.1.1: signal/noise ratio (dB) versus N

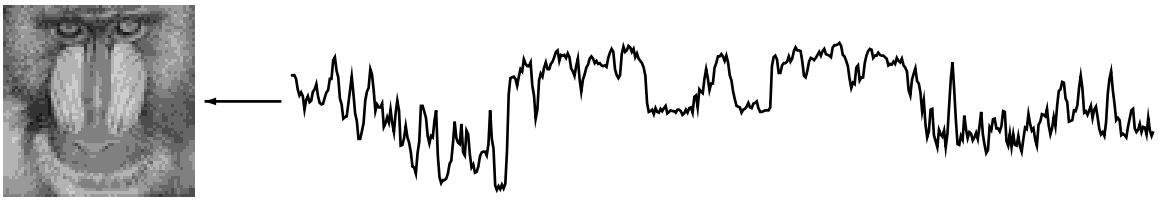


Figure 5.1.2: scanline function from mandrill image

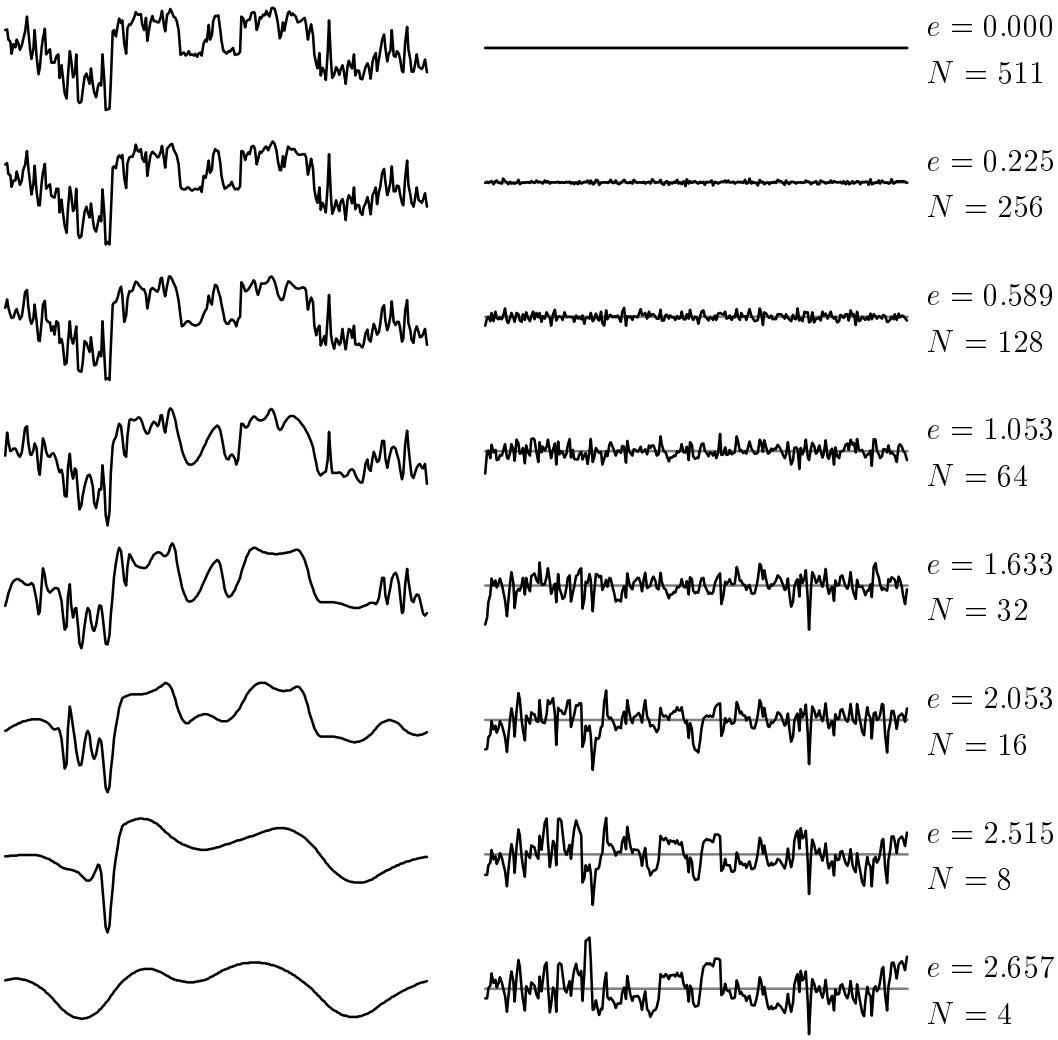


Figure 5.1.3: approximations and errors

The bottom-up approximation process just described works well when a uniform set of positions $P_{\ell_{\max}}$ are given as input, and when only a small fraction of the compacted displacements are set to zero. If a large fraction of the compacted displacements are set to zero, then the process is inefficient in the sense that too much input data is provided. Furthermore, for continuous input functions the process may be intractable for high accuracies because the required number of uniformly-spaced input positions grows large. This may hold even when the number of resulting compacted displacements is small: consider as an example a tall, narrow spike approximated with respect to the supremum norm.

5.2 Top-Down Approximation

The alternative to bottom-up approximation is a top-down approximation algorithm. The top-down approach adds new nonzero compacted displacements instead of removing them. Thus the computational work and data space is linear with respect to the number of nonzero compacted displacements in the output. Another way to view these different approaches is in terms of the accuracy/complexity tradeoff curve. The bottom-up process attempts to follow the optimal tradeoff curve while decreasing the complexity. The top-down process attempts to follow the optimal tradeoff curve while increasing the complexity.

The top-down approximation algorithm is more complicated than the bottom-up algorithm. The additional difficulty arises because the approximation process must be based on inexact or "fuzzy" knowledge of the target function $g(t)$. This requires that (a) local estimates of $g(t)$ are readily available, and that (b) the positions $P_{\ell,i}$ can be quickly fit to the estimated version of g. The advantage to the top-down approximation process is that a wide variety of target functions can be approximated directly without the intermediate conversion to high-resolution B-splines that is assumed for bottom-up fitting.

The next section describes how $g(t)$ can be estimated locally, and how bounds on these estimates can be obtained. The section following that gives a fast method for computing the ideal fit for positions $P_{\ell,i}$ with respect to the estimated $g(t)$. In section 5.2.3, these estimation and fitting tools are used to generate compacted displacements in a way that attempts to track the optimal accuracy/complexity tradeoff curve. Section 5.2.4 gives a demonstration of this top-down approximation algorithm at work.

5.2.1 Local Estimates

The top-down approximation process requires that the target function $g(t)$ be estimated over domain partitions made up of bintree intervals $I_{\ell,i}$. It is assumed that

the estimate $\tilde{g}(t)$ is of the form

$$\tilde{g}(t) = \sum_j G_j B_j(t)$$

where $B_j(t)$ are basis functions and $G_j \in \Re$ are constants that give a reasonable estimate of $g(t)$. Each basis function $B_j(t)$ should have a corresponding domain interval I_{ℓ_j,i_j}, and should be smooth inside this interval and zero outside.

For efficiency of the preprocessing in the next section, the basis functions should be translated and dilated copies of a small set of canonical basis functions. A good choice for the canonical basis functions are the Bernstein/Bézier basis functions for small degrees [15]. There is no concern about maintaining continuity between domain intervals, since the estimate will be used over each domain interval independently and the output approximation will be smooth regardless of the estimate continuity.

For degree n, the Bernstein/Bézier basis functions on $I = [0, 1]$ are

$$\lambda^s(t) = \binom{n}{s} (1-t)^{n-s} t^s$$

for $s = 0, 1, \ldots, n$. For a bintree interval $I_{\ell,i}$, the translated and dilated basis functions are

$$\lambda^s_{\ell,i}(t) = \lambda^s(2^\ell t - i)$$

The final piece of information needed for top-down approximation is a bound on the error of the local estimate $\tilde{g}(t)$ over a domain interval $I_{\ell,i}$:

$$|g(t) - \tilde{g}(t)| < E_{\ell,i}$$

for all $t \in I_{\ell,i}$.

The method of determining these local estimates and bounds will vary from one application to the next. Some target functions $g(t)$ can provide this information easily: examples are B-splines with general knot sequences (see section 2.2.1 or [15]), and functions that are computed using interval arithmetic [31]. If a function can be sampled for both positions and derivatives, and its derivative bounds are known (either locally or globally), a local estimate and bound can be formed. In the worst case, if only samples can be taken, then a statistical estimate and bound can be obtained.

An example target function is shown in Figure 5.2.1. Three piecewise estimates are shown for Bernstein/Bézier degrees $n = 0, 1, 2$. In each case, the estimate pieces are split until each error bound is within a specified tolerance. The estimates and bounds were produced using interval arithmetic in these examples.

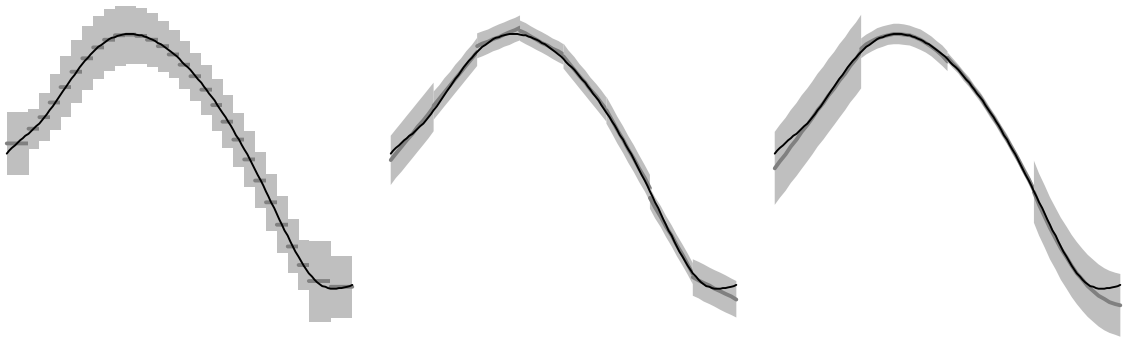


Figure 5.2.1: piecewise estimates and bounds
(a) degree $n = 0$ (b) degree $n = 1$ (c) degree $n = 2$

5.2.2 Ideal Fit of the Estimate

In the bottom-up fit-and-compact process, it was assumed that high-resolution positions $P_{\ell_{\max}}$ were available at level $\ell_{\max}$. Each level above $\ell_{\max}$ was fit as

$$P_\ell = \mathbf{F}^{\ell_{\max}-\ell} P_{\ell_{\max}}$$

Ideally, the fit positions P_ℓ should be determined based on exact knowledge of the target function $g(t)$. This is equivalent to saying that the ideal fit positions should be determined by letting $\ell_{\max}$ go to infinity

$$P_{\ell,i} = \lim_{\ell_{\max}\to\infty} \left[\mathbf{F}^{\ell_{\max}-\ell} \hat{P}_{\ell_{\max}}\right]_i$$

where

$$\hat{P}_{\ell_{\max},j} = g(2^{-\ell_{\max}} j)$$

Let the limit-fitting operation for level ℓ be denoted

$$P_\ell = \mathbf{F}^{\ell,\infty} g$$

Attempting to compute these limit fit positions directly for $g(t)$ is impractical, but the situation is simpler when piecewise estimates of $g(t)$ are used.

Suppose that Bernstein/Bézier estimates are available for $g(t)$ over all bintree domain intervals $I_{\ell,i}$:

$$\tilde{g}_{\ell,i}(t) = \sum_s G^s_{\ell,i} \lambda^s_{\ell,i}(t)$$

Define the initial estimate of level ℓ positions based on the level ℓ estimate pieces

$$\begin{aligned} P^0_{\ell,i} &= \mathbf{F}^{\ell,\infty}_i \tilde{g}_\ell \\ &= \mathbf{F}^{\ell,\infty}_i \textstyle\sum_{j,s} G^s_{\ell,j} \lambda^s_{\ell,j} \\ &= \textstyle\sum_{j,s} G^s_{\ell,j} (\mathbf{F}^{\ell,\infty}_i \lambda^s_{\ell,j}) \end{aligned}$$

As discussed in the next section, the values $\mathbf{F}_i^{\ell,\infty}\lambda_{\ell,j}^s$ may be precomputed efficiently. The initial estimate fit process may be summarized by introducing an operator $\mathbf{B}$ so that

$$P_\ell^0 = \mathbf{B}G_\ell$$

The level ℓ positions may be estimated based on successively finer estimates of g as

$$P_\ell^m = \mathbf{F}^m\mathbf{B}G_{\ell+m}$$

In the top-down approximation algorithm described in section 5.2.3, the estimate fit operator $\mathbf{B}$ will be used locally so that incremental updates to the approximation are computed quickly.

Precomputing the Estimate-Fit Kernel

Since the local estimate bases are translated and dilated copies of the basis for domain $I = [0, 1]$, the constants $\mathbf{F}_i^{\ell,\infty}\lambda_{\ell,j}^s$ do not vary based on level or position, but rather on the estimate basis index s and on the relative neighborhood index $j - i$. Let the constants be called β_{j-i}^s where

$$\beta_{j-i}^s = \mathbf{F}_i^{\ell,\infty}\lambda_{\ell,j}^s$$

for any ℓ, i and j. The estimate-fit operator $\mathbf{B}$ is made up of these constants that form its kernel.

These β values may be precomputed for a given nth-degree estimate basis λ and limit-fit operator $\mathbf{F}^{\ell,\infty}$. For finite-width fit operators $\mathbf{F}$, only a small neighborhood of β's are nonzero (the neighborhood is no larger than the fit operator kernel width).

For many applications it is not critical that this computation be performed quickly, since the estimate-fit kernel β_{j-i}^s can be precomputed. However, a naive algorithm to compute β would cause storage and computation time to double each time $\ell_{\max}$ increases by one in the limit process defining β. This becomes intractable if β is to be computed to high numerical precision. Fortunately, a simple feedback algorithm can be developed based on the subdivision properties of the Bernstein/Bézier estimate bases. This feedback algorithm will require a small, constant amount of time and space to increase $\ell_{\max}$ by one in the limit process.

For the degree-n Bernstein/Bézier estimate basis λ, the left and right halves of each basis function are weighted sums of the same basis functions dilated and translated as

$$\lambda^s(t) = \left(\sum_{s'} a_{s,s'}\lambda^{s'}(2t)\right) + \left(\sum_{s'} b_{s,s'}\lambda^{s'}(2t-1)\right)$$

where $a_{s,s'}$ and $b_{s,s'}$ can be computed using the de Casteljau algorithm [15]. The splitting of the degree $n = 2$ Bernstein/Bézier estimate basis is depicted in Figure 5.2.2.

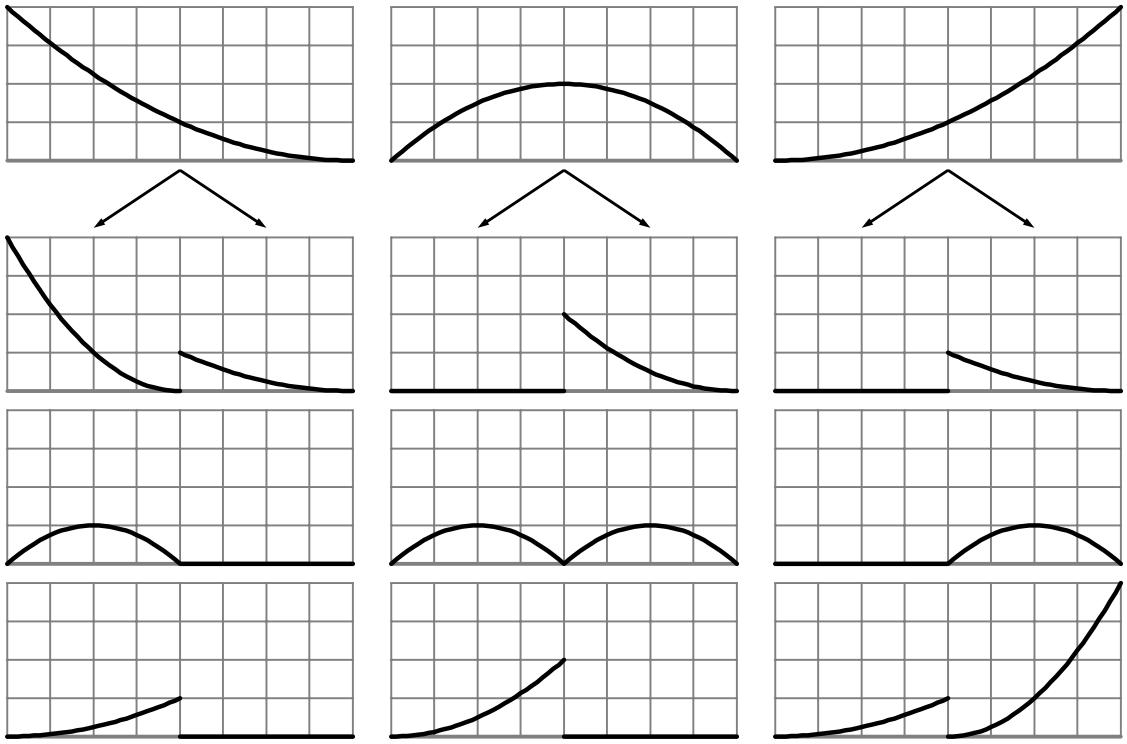


Figure 5.2.2: estimate basis splitting

The feedback algorithm for computing β starts with single samples from the estimate basis functions

$$\beta_i^{s,0} = \begin{cases} \lambda^s(0) & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

Now the limit computation of β may be iterated as

$$\beta^{s,m+1} = \mathbf{RF}A^{s,m}$$

where

$$A_i^{s,m} = \sum_{s'} \left(a_{s,s'} \beta_{0-i}^{s',m} + b_{s,s'} \beta_{1-i}^{s',m} \right)$$

and $\mathbf{R}$ is the sequence-reversal operator

$$\mathbf{R}_{i,j} = \begin{cases} 1 & \text{if } j - i = 0 \\ 0 & \text{otherwise} \end{cases}$$

The estimate-fit kernels are shown in Figure 5.2.3. The fit operators $\mathbf{F}$, shown along the vertical axis of the figure, are for degrees $n = 1, 2, 3$ and respective width parameters $k = 1, 1, 2$. Various Bernstein/Bézier basis functions are shown on the horizontal axis.

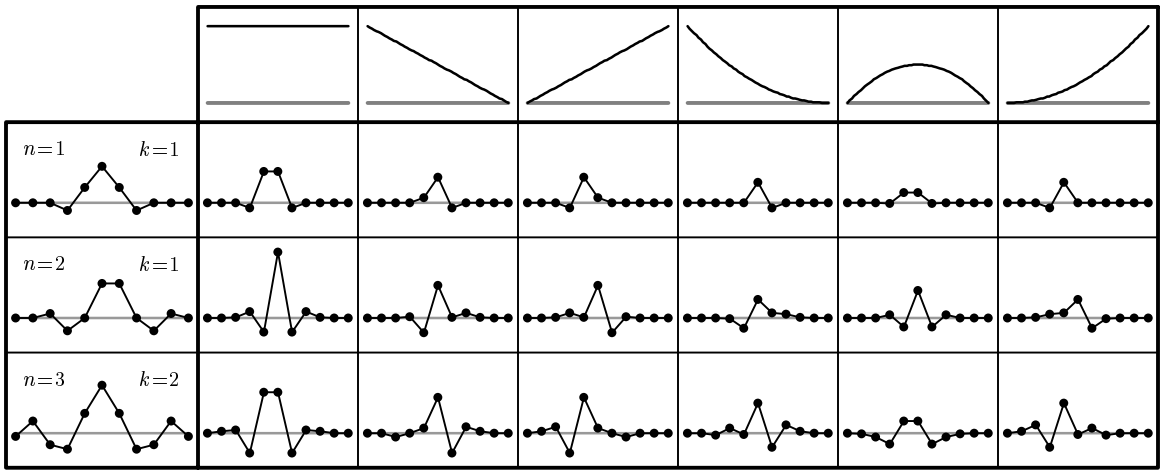


Figure 5.2.3: estimate-fit kernels

It should be noted that the limit process is not well defined for small filter widths. Numerical tests give convergence of the β computation as follows for all Bernstein/Bézier bases up to degree nine. Figure 5.2.4 lists the minimum filter width parameter k required for convergence for fit filter degrees $n = 0, \ldots, 9$.

Filter Degree	Minimum Width	Filter Degree	Minimum Width
$n = 0$	$k = 0$	$n = 5$	$k = 5$
$n = 1$	$k = 0$	$n = 6$	$k = 7$
$n = 2$	$k = 1$	$n = 7$	$k = 11$
$n = 3$	$k = 1$	$n = 8$	$k = 15$
$n = 4$	$k = 2$	$n = 9$	$k = 19$

Figure 5.2.4: minimum filter widths for β convergence
for various filter degrees

5.2.3 Top-Down Algorithm

The previous sections have introduced the tools needed to perform top-down approximation on a target function $g(t)$. Local estimates of g must be available for each bintree domain interval $I_{\ell,i}$ in Bernstein/Bézier form:

$$\tilde{g}_{\ell,i}(t) = \sum_s G^s_{\ell,i} \lambda^s_{\ell,i}(t)$$

These estimates have error bounds $E_{\ell,i}$ such that

$$|g(t) - \tilde{g}(t)| < E_{\ell,i}$$

for all $t \in I_{\ell,i}$. The limit-fit positions $\tilde{P}_{\ell,i}$ are computed using using the precomputed $\mathbf{B}$ operator

$$\tilde{P}_\ell = \mathbf{B}G_\ell$$

For a single entry in $\tilde{P}_\ell$ this becomes

$$\tilde{P}_{\ell,i} = \sum_{j,s} G^s_{\ell,j}\beta^s_{j-i}$$

The top-down algorithm maintains a list of bintree domain intervals whose compacted displacements influence a neighborhood that is out of tolerance. The algorithm proceeds by splitting each of these interval list entries, adding a new compacted displacement at each new interval, and deleting any interval list entries whose neighborhood of influence has come into tolerance. This processing on the list is performed in phases for efficiency (these phases are described later in this section). The phases are repeated until the dyadic-spline approximation $f(t)$ has come within tolerance (i.e. the interval list has become empty), or until the number of compacted displacements has exceeded a threshold.

The top-down approximation algorithm is depicted in Figure 5.2.5. The target function $g(t)$ is shown in part (a). It is the sum of two "bumps," one wide and one narrow, which are each transcendental functions of the form $hb((t-t_c)/w)$ where

$$b(t) = \begin{cases} e^{-\tan^2(\frac{\pi}{2}t)} & \text{if } t \in (-1,1) \\ 0 & \text{otherwise} \end{cases}$$

These bump functions have closed forms for their derivatives of various orders. It is straightforward to evaluate local estimates and bounds using interval arithmetic [31].

The initial interval list has one entry, as shown in part (b) of the figure. The dyadic-spline representation at this point has one displacement $D_{0,0}$ at level zero, and one compacted displacement $Q_{0,0}$. The first list-processing cycle splits this interval in two, and adds displacements for each of the two new child intervals, $Q_{1,0}$ and $Q_{1,1}$. The old compacted displacement has been updated based on the improved knowledge of the target function. The result after this first cycle is shown in part (c). Parts (d), (e) and (f) show the results of subsequent list-processing cycles. Notice that domain intervals that are away from the narrow bump come into tolerance in the early cycles and no longer require consideration.

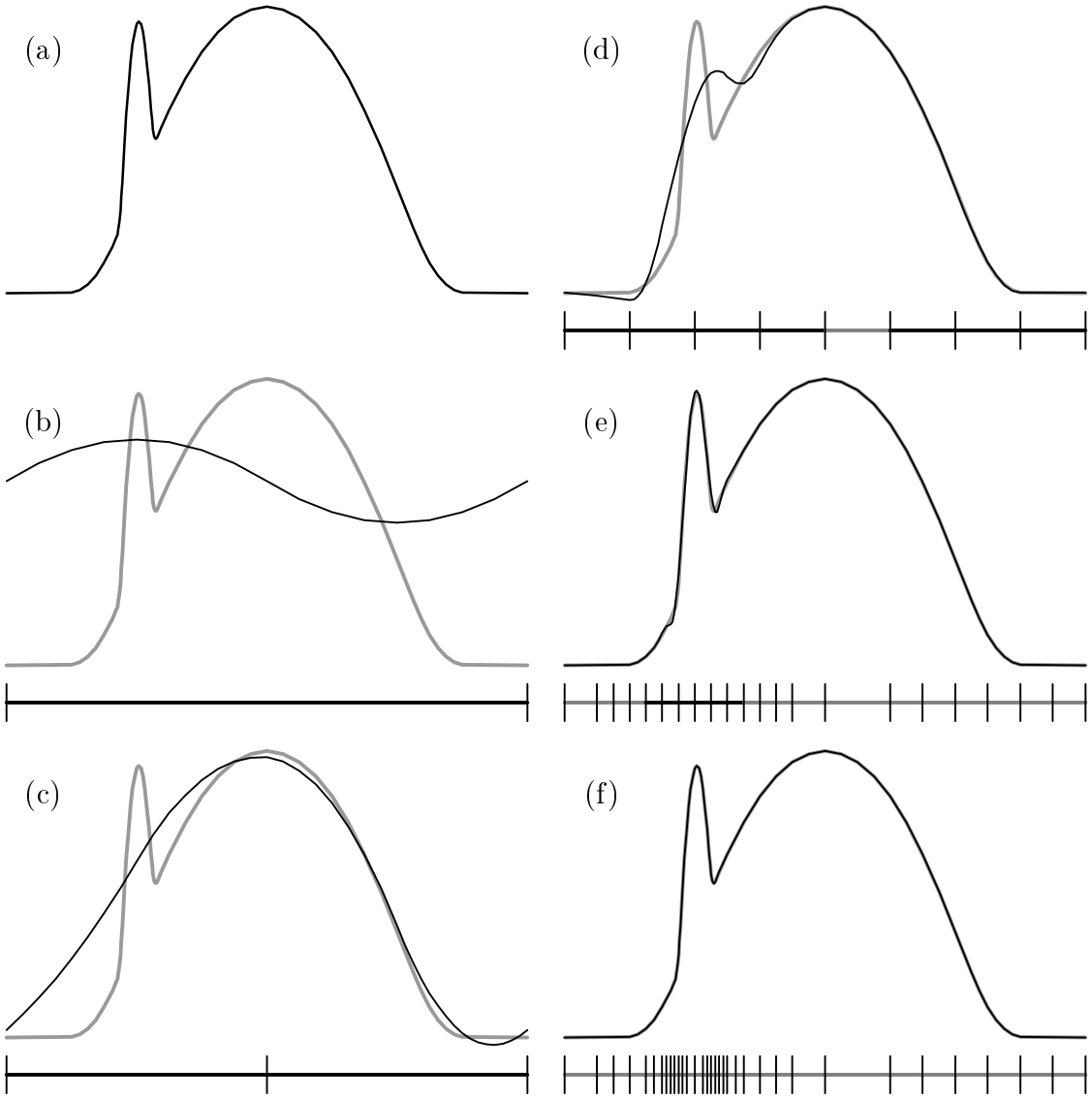


Figure 5.2.5: top-down approximation cycles

The list-processing cycle is divided into phases as follows:

interval split: split each interval-list entry into its two children

The interval-split phase replaces each interval-list entry $I_{\ell,i}$ with entries for its two children, $I_{\ell+1,2i}$ and $I_{\ell+1,2i+1}$. The compacted displacements associated with these new intervals, $Q_{\ell+1,2i}$ and $Q_{\ell+1,2i+1}$, are added to the active compacted displacements. The remaining phases will initialize these newly-active compacted displacements, update the dyadic-spline output approximation $f(t)$ and toss out any interval-list entries that have come into tolerance.

fit estimate: compute local estimates and limit-fit positions

After deciding on the locations of the new compacted displacements, compute the limit-fit positions $\tilde{P}_{\ell,i}$ that they depend on. Some additional local estimates $G^s_{\ell,i}$ and $E_{\ell,i}$ may need to be computed as well. These computations are stored and then reused when needed again. The old limit-fit positions are updated if the lower-level positions that they may be fit from are available and have changed. This completes the fit-estimate phase of the top-down approximation cycle.

compacted displacements: update or initialize compacted displacements

Next comes the compacted-displacements phase, wherein the limit-fit positions just computed are used. A current compacted displacement is updated if any of the limit-fit positions it depends on have changed or if the compacted displacement is newly active. Note that old compacted displacements tend to be improved as better knowledge of the target function becomes available. This new knowledge is acquired through the lower-level updates and subsequent fitting.

approximation: update output approximation's displacements and positions

Given the updated active compacted displacements, the approximation phase updates the dyadic-spline output function $f(t)$. The displacements and positions of $f(t)$, $D_{\ell,i}$ and $P_{\ell,i}$, are updated based on the changes to the compacted displacements $Q_{\ell,i}$. The output positions are then used to determine an error bound for each domain interval near an entry in the interval list. "Nearness" here means that the domain interval is within the neighborhood of influence of the compacted displacements rooted at some entry in the interval list. Determining these neighborhoods of influence is similar to determining the influence of a single displacement, as discussed in section 3.2. Note that the only output displacements and positions that need to be computed are those that influence the error bounds of interest.

interval toss: remove interval-list entries that are in tolerance

The last part of the top-down approximation cycle is the interval-toss phase. For each interval-list entry, a neighborhood of intervals is tested to see if it is in tolerance. Any interval-list entry whose neighborhood is in tolerance is deleted. The neighborhood consists of those intervals that are influenced by the compacted displacements rooted at the interval-list entry. The tolerance test determines whether the output approximation is guaranteed to differ by less than a certain bound from the local estimate for an interval. Since the local estimates are in Bernstein/Bézier form, it is appropriate to convert the local approximation to this form as well and raise the degree of either local representation so that they are easily compared. The local approximation may be obtained as a B-spline

approximation as described earlier in section 3.2. These B-spline approximations may be converted to Bernstein/Bézier form by blossom evaluations on the Bernstein/Bézier knots. Degree-raising is also straightforward using blossoming [32].

5.2.4 Top-Down Approximation Results

For an illustrative example, the top-down fit is applied to a target function that is again a sum of transcendental bumps. The target function is shown in Figure 5.2.6, along with several approximations of increasing accuracy. Magnified views are shown of the function in a neighborhood that contains some fine detail. The domain decomposition is shown for each approximation. Notice that the computational effort focuses on the neighborhood containing the fine details. This is significantly more efficient that the bottom-up approximation process. The gains in efficiency increase as the output quality increases. Of course these efficiencies depend on the fact that the output compacted displacements are sparse, and on the fact that the target function is available top-down as local estimates and not just as uniformly-sampled data.

The next figure depicts the accuracy/complexity tradeoff curve produced by the top-down approximation process. The target function is the one used above. The dyadic spline has degree $n = 2$ and filter-width parameter $k = 1$. The local estimates are in quadratic Bernstein/Bézier form. The graph shows the logarithm of the reciprocal of the maximum absolute error (in decibels) as a function of the number of nonzero compacted displacements. The black curve is for the top-down algorithm. For comparison, the gray curve depicts the tradeoff for the bottom-up algorithm. The bottom-up algorithm has the advantage that it is working from detailed knowledge of the target function instead of the local estimates used by the top-down algorithm. An area of future research is to improve the top-down tradeoff curve and bring it closer to the bottom-up one.

Throughout this discussion on top-down approximation, it has been assumed that the approximation error is measured with respect to the supremum norm. In other words, the top-down algorithm was described in terms of bounds on the maximum pointwise deviation from the target function. A tolerance in this sense was an upper bound on acceptable maximum deviations. Other norms are guaranteed to have global tolerances met by setting the supremum-norm tolerance appropriately. However, the decisions on where to subdivide should be tuned to the norm desired. Effective means of doing this are an area of future research.

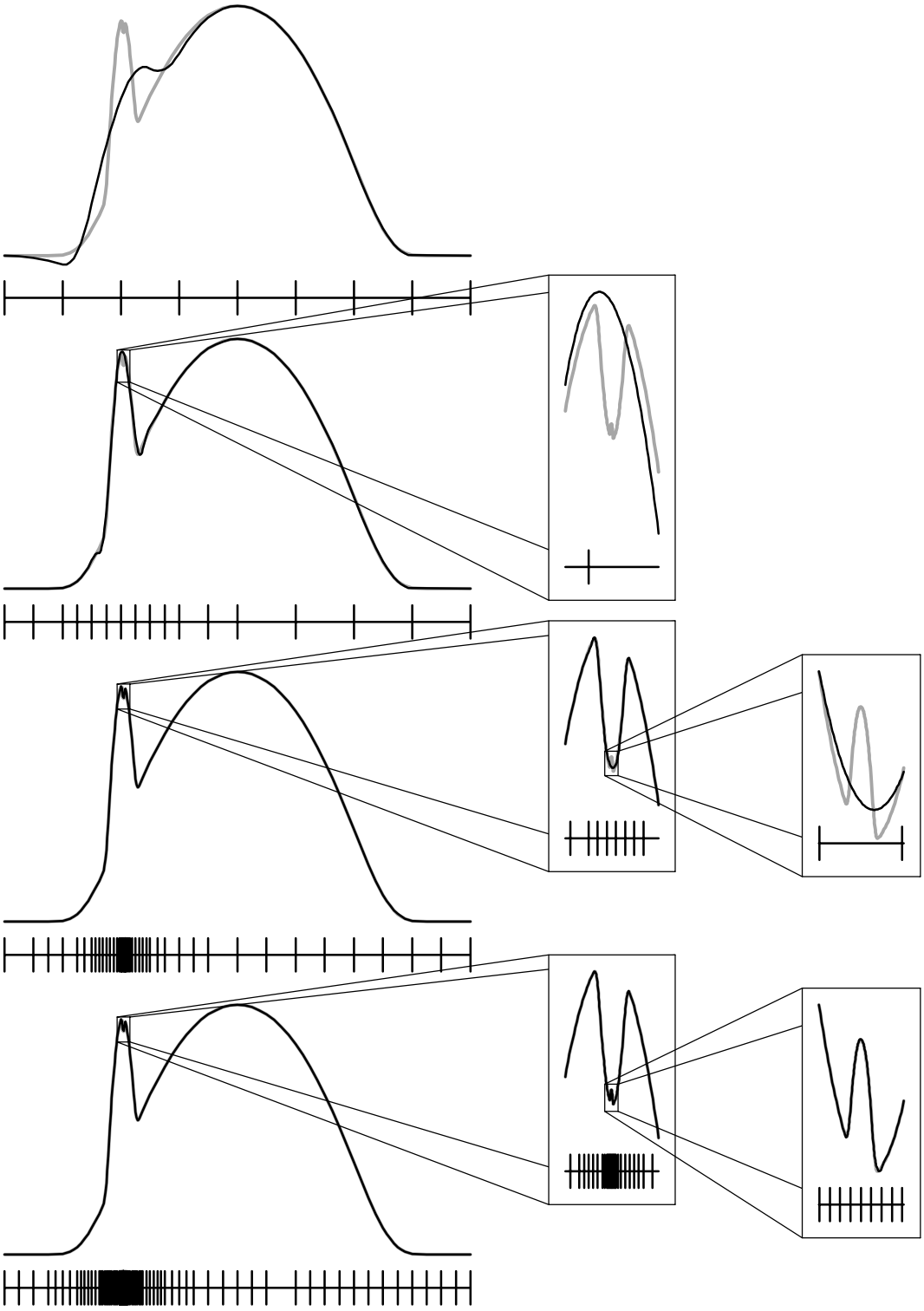


Figure 5.2.6: top-down approximations

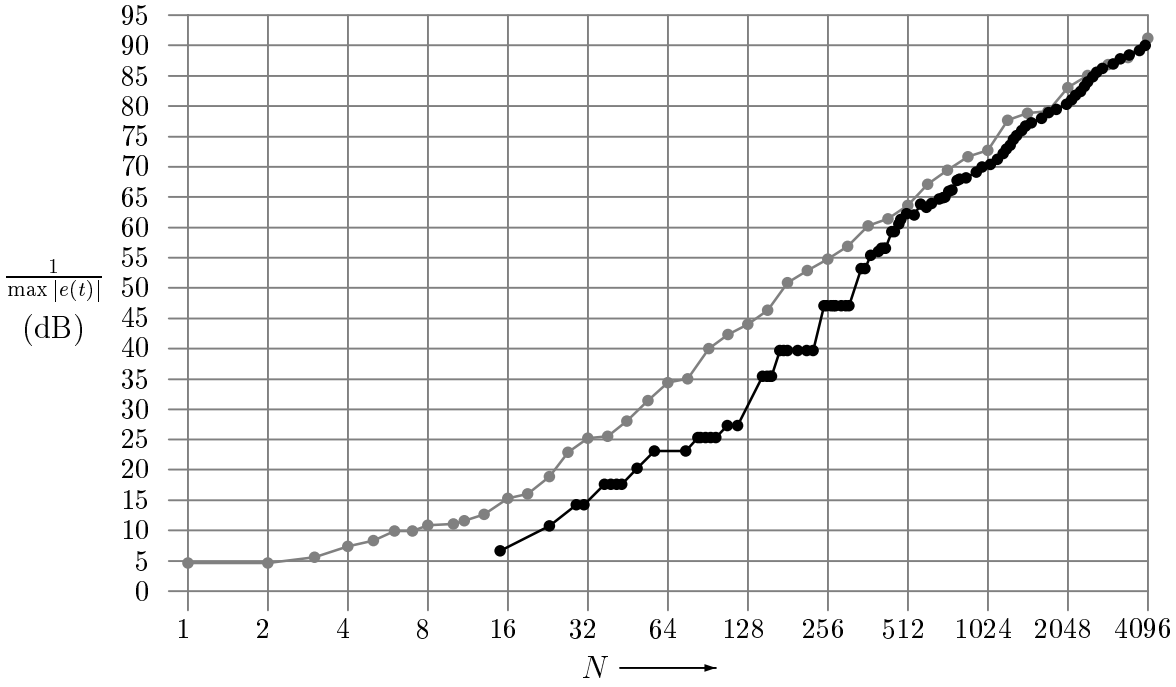


Figure 5.2.7: accuracy/complexity tradeoff curves
(black) top-down (gray) bottom-up

Chapter 6

Curve Design

The discussions of synthesis, analysis and approximation using dyadic splines have tacitly assumed that the displacements are known in advance, or that they have been obtained through an approximation process on a known target function. This chapter uses the low-level synthesis, analysis and approximation tools as building blocks so that new functions may be designed in an interactive setting. The goal is to provide the user with several different mechanisms for manipulating functions that take advantage of the multiresolution displacement representation.

Six design mechanisms will be discussed, some of which are variants on methods introduced by other researchers, now applied in the dyadic-spline framework. These are depicted in Figure 6.0.1. The simplest curve modification is achieved by directly editing the displacements, as shown in part (a). Two complimentary types of modifications are shown in parts (b) and (c). Part (b) shows the effect of smoothing the displacements in a neighborhood, while (c) shows the effect of roughening. Global smoothing for curves was introduced by Finkelstein and Salesin in [16]. Roughening is an application of fractals [27]. Part (d) shows *offset-frame displacements* of the type introduced by Forsey and Bartels [18]. This gives displacements a local coordinate frame derived from the underlying curve, so that the displacement tracks local and global changes in shape. The next mechanism is *template control*, shown in part (e). This generalizes the notion of a displacement edit to allow *templates* to be superimposed at arbitrary positions and scales on the underlying curve. One form of template is related to the *pasting* method introduced by Barghiel in her Master's thesis [1]. The final mechanism shown in part (f) is *sculpting*, wherein a tool modifies a curve continuously as it moves along a user-specified path. Interactive sculpting (of surfaces) was introduced by Stoneking in his Master's thesis [36]. Each of these editing mechanisms fits nicely into the framework of dyadic splines. The remainder of this chapter discusses each in turn.

So far, the functions under discussion have had a single real parameter $t \in \Re$, and have evaluated to a single real value $f(t) \in \Re$. This chapter will continue to rely on a single real parameter t, but now the evaluation of $f(t)$ will give a point on the Cartesian plane:

$$f(t) = \begin{bmatrix} f_x(t) \\ f_y(t) \end{bmatrix}$$

where $f_x(t)$ and $f_y(t)$ are real-valued functions. In general, only a finite interval of parameters are of interest. To simplify the discussion, assume this interval is $t \in [0, 1]$.

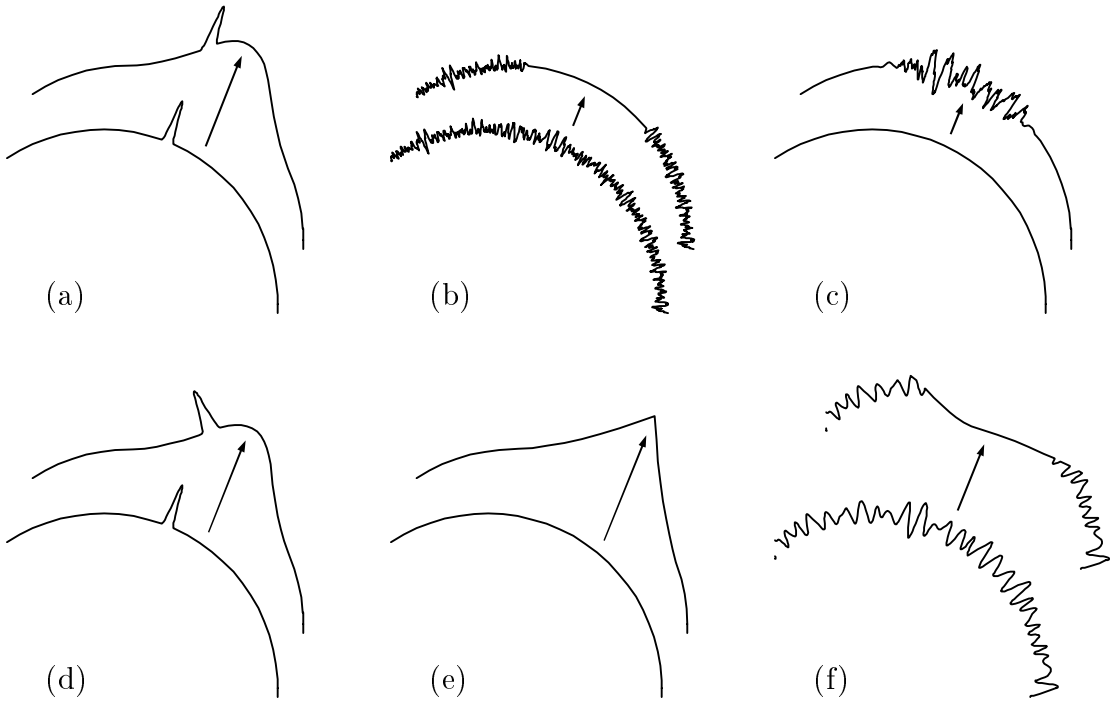


Figure 6.0.1: curve design mechanisms

6.1 Displacement Edits

Since dyadic splines are defined in terms of the displacements $D_{\ell,i}$, the most direct means of manipulating a dyadic-spline curve is to interactively "drag" one displacement at a time. The choice of level ℓ determines the width of the "bump" produced, while the choice of i determines the location of the bump. Two different bump manipulations are shown in Figure 6.1.1, at two different scales and positions.

A dramatic difference between dyadic-spline displacement edits and conventional B-spline control-point edits is that the displacement control handles exist at multiple resolutions simultaneously. This is shown in Figure 6.1.2. When a coarse-resolution displacement is moved, the spike position tracks the coarse-resolution effect. This occurs because the finer displacement that defines the spike shape is superimposed on the coarse underlying shape. Unfortunately the spike displacement doesn't track the *orientation* of the coarse underlying shape. This is solved in section 6.3 through the use of offset coordinate frames.

Figure 6.1.1: displacement edits

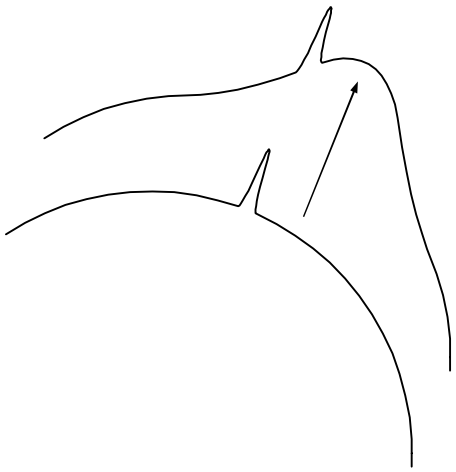


Figure 6.1.2: superposition of detail

A disadvantage of direct displacement edits is that the user is forced to be aware of the structure of the representation making up the curve. It is more intuitive to have the curve behave the same for continuous variations in edit position and width. Also, the shape of the edit effect should not be determined by the underlying basis functions, but should be allowed to vary to suit a particular application. These enhancements are provided by template edits, as discussed in section 6.5.

6.2 Neighborhood Smoothing

There are several "underlying" shapes that may be readily extracted from the dyadic-spline curve. One family of shapes may be obtained by changing a *smoothing*

parameter ℓ_s continuously from 0 to $\ell_{\max}$, where $\ell_{\max}$ is the level of maximum detail. The smoothed function $\bar{f}$ with displacements $\bar{D}_{\ell,i}$ is obtained from function f with displacements $D_{\ell,i}$ as follows:

$$\bar{D}_{\ell,i} = \begin{cases} D_{\ell,i} & \text{if } \ell < \ell_s \\ (\ell_s - (\ell - 1))D_{\ell,i} & \text{if } \ell - 1 < \ell_s \leq \ell \\ 0 & \text{otherwise } (\ell_s \leq \ell - 1) \end{cases}$$

This smoothing operation works best when the displacements are in canonical form, as discussed in section 4.4.1. An example of the resulting family of curves is shown in Figure 6.2.1.

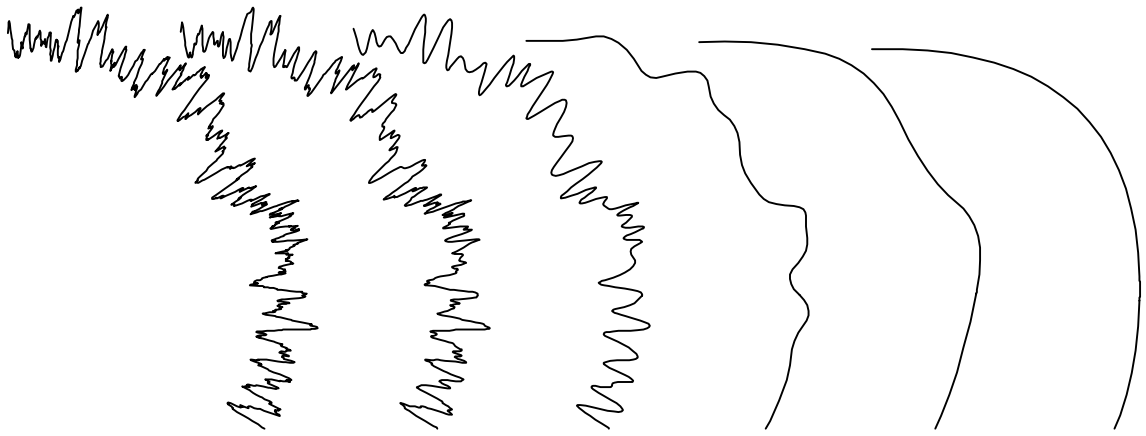


Figure 6.2.1: curve smoothing

For purposes of editing, smoothing should be allowed to apply to an arbitrary segment of the curve, and the effect of smoothing should taper off near the ends of this segment. A simple mechanism for achieving this is to blend between the original and smooth displacements based on the overlap of the smoothing segment with the displacement's neighborhood of influence. Let $I_s = [t_{s0}, t_{s1}]$ be the domain interval being smoothed. For a degree n displacement $D_{\ell,i}$, the domain interval that it influences is

$$I_D = [t_{D0}, t_{D1}] = \left[2^{-\ell}(i - \lfloor (n+1)/2 \rfloor), 2^{-\ell}(i + \lfloor (n+2)/2 \rfloor)\right]$$

Let $t_0 = \max\{t_{s0}, t_{D0}\}$ and $t_1 = \min\{t_{s1}, t_{D1}\}$. The fraction of I_D that overlaps I_s is

$$q = \begin{cases} \frac{t_1 - t_0}{t_{D1} - t_{D0}} & \text{if } t_1 > t_0 \\ 0 & \text{otherwise} \end{cases}$$

This fraction will be used as a blend factor between the smoothed and original displacements. Figure 6.2.2 depicts the fraction of overlap q for example intervals I_s and I_D in the domain.

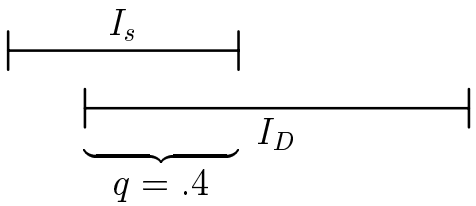


Figure 6.2.2: domain overlap as blend factor

Now the locally-smoothed version of $D_{\ell,i}$ is defined by

$$\breve{D}_{\ell,i} = (1-q)D_{\ell,i} + q\bar{D}_{\ell,i}$$

An example of local smoothing is shown in Figure 6.2.3.

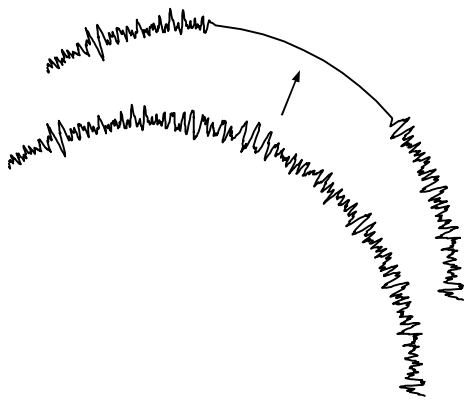


Figure 6.2.3: local smoothing

6.3 Offset Displacements

As mentioned earlier, one problem with directly editing displacements is that the orientation of the detail does not track changes to the coarse shape. This is solved by specifying offsets relative to a local coordinate frame obtained from the tangent and normal vectors of a coarser (i.e. smoothed) version of the curve. More formally, the user specifies offset displacement $\hat{D}_{\ell,i}$, and the displacement $D_{\ell,i}$ is obtained as

$$D_{\ell,i} = A_{\ell,i}\hat{D}_{\ell,i}$$

where

$$A_{\ell,i} = \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix}$$

and u is a unit tangent, and v is the normal obtained by rotating u counterclockwise by 90 degrees. The tangent u is taken with respect to the smoothed curve $\bar{f}(t_m)$ for a chosen $\ell_s \leq \ell - 1$ and $t_m = 2^{-\ell}(i + ((n+1) \bmod 2)/2)$. The domain position t_m gives the maxima of the basis function associated with $D_{\ell,i}$.

The effect of this formulation is shown in Figure 6.3.1. Notice how the spike now tracks the position *and* orientation of the underlying curve, in contrast to to the direct displacement edits of Figure 6.1.2.

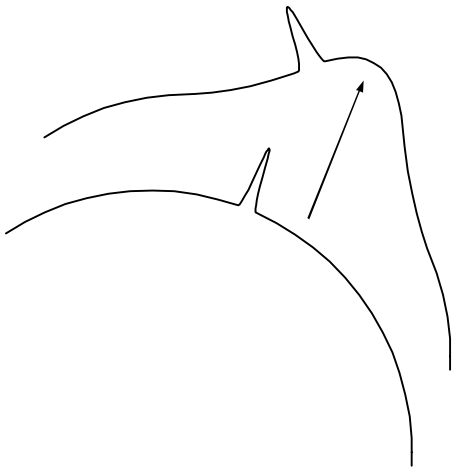


Figure 6.3.1: offset-frame edits

Typically $\ell_s = \ell - 1$ is used in simple offset-frame formulations for displacements at level ℓ, and t_m is determined automatically. A generalized offset-frame formulation is obtained by letting ℓ_s and t_m be specified arbitrarily for each level ℓ and index i (but with the constraint $\ell_s \leq \ell - 1$).

6.4 Neighborhood Roughening

A complimentary operation to smoothing is *roughening*. Although many types of detail might be added to a smooth curve, such as ripples and loops, this section will focus on generic "rough" detail generated automatically by a fractal construction.

Let the resolutions of roughness be determined by two limits ℓ_{r0} and ℓ_{r1} that vary continuously. Fractal detail is generated by first producing random compacted displacements for levels $\ell = \lfloor \ell_{r0} \rfloor, \ldots, \lceil \ell_{r1} \rceil$ as

$$\tilde{Q}_{\ell,i} = b_\ell h 2^{-l} R_{\ell,i}$$

where $R_{\ell,i}$ is chosen uniformly from $[-1,1]$ and h is a scaling factor. The factor b_ℓ puts into effect the continuous variation of the resolution limits, and is defined as

$$b_\ell = \begin{cases} 1 & \text{if } \ell_{r0} \le \ell \le \ell_{r1} \\ \ell + 1 - \ell_{r0} & \text{if } \ell < \ell_{r0} < \ell + 1 \\ \ell_{r1} - (\ell - 1) & \text{if } \ell - 1 < \ell_{r1} < \ell \\ 0 & \text{otherwise} \end{cases}$$

The expanded versions of these compacted displacements are added to the smoothed displacments of the original curve, relative to the local normals of the smoothed curve as described in the last section. This effectively replaces any detail on the curve with the fractal detail. An appropriate smoothing parameter is $\ell_s = \ell_{r0}$.

Roughening is depicted in Figure 6.4.1 for various combinations of ℓ_{r0} and ℓ_{r1}.

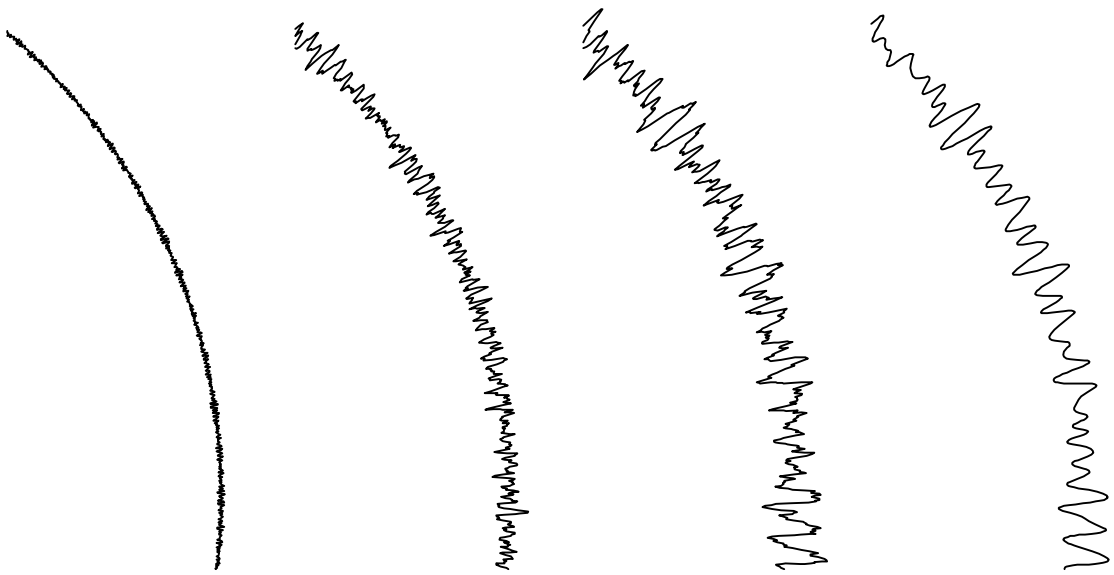


Figure 6.4.1: curve roughening

Local roughening is put into effect in the same manner as local smoothing. Let $I_r = [t_{r0}, t_{r1}]$ be the domain interval being roughened, and let $D_{\ell,i}$ be an original displacement. Let q be defined as in section 6.2, giving the fraction of $D_{\ell,i}$'s interval of influence I_D that overlaps I_r. Then the locally-roughened displacement is defined as

$$\check{D}_{\ell,i} = (1-q)D_{\ell,i} + q\tilde{D}_{\ell,i}$$

where $\tilde{D}_\ell = \mathbf{E}\tilde{Q}_{\ell-1}$ are the expanded versions of the random compacted displacements ($\mathbf{E}$ is the expansion operator of the dyadic-spline filter bank; see section 4.2).

An example of local roughening is shown in Figure 6.4.2.

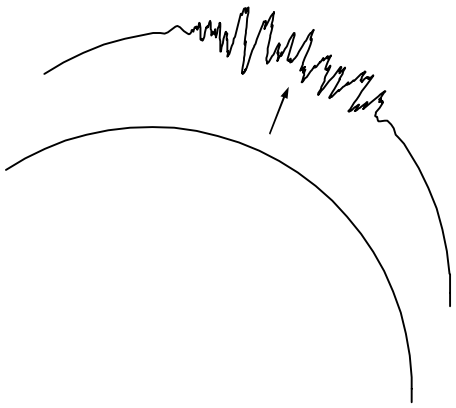


Figure 6.4.2: local roughening

6.5 Template Edits

As pointed out in Barghiel's thesis [1], the use of offset frames for displacements is just an approximation of the more general notion of continuously varying offsets with general positions, scales and shapes. Generalizing further, the fundamental capability is to allow arbitrary *template* shapes to be superimposed on an original curve. Additional examination of the method reveals options which can be used to vary the template effect.

The base template operation simulates the effect of adding *generalized basis functions* to the curve $f(t)$. Such functions are defined in terms of a real-valued template function $g(s)$ that is positioned using a one-to-one mapping $h(s)$ from s to t:

$$G(t) = g(h^{-1}(t))$$

As with traditional basis functions, it is applied to an initial curve function $f(t)$ by scaling and addition:

$$\hat{f}(t) = f(t) + cG(t)$$

Here c is a control vector specified by the user as a design handle (since $G(t)$ is scalar valued like conventional basis functions, c must be a vector). The user can also select or modify the template $g(s)$ and the template positioning function $h(s)$. The vector c may be optionally derived from a local offset-frame coordinate vector $\hat{c}$ as $c = A\hat{c}$. In this case, the offset frame A is obtained in the same manner as for offset-frame displacements, as discussed in section 6.3.

One of the benefits of this type of template edit is that arbitrary sections of a curve may be manipulated without regard to the structure of the curve representation. The curve will effectively behave as though it had an infinite variety of basis

functions, where the user can pick the basis function interactively to get a desired effect. Three examples are depicted in Figure 6.5.1. The first template allows a specific point on the curve to be dragged so that the point exactly tracks (i.e. interpolates) the drag point. The curve remains smooth around the point, and no extraneous oscillations are introduced. This is similar to typical dyadic-spline basis functions, but with continuous variations in domain position and width. The second template gives interpolation of the drag point, but also introduces a corner. The third template exhibits the generality of the template shapes.

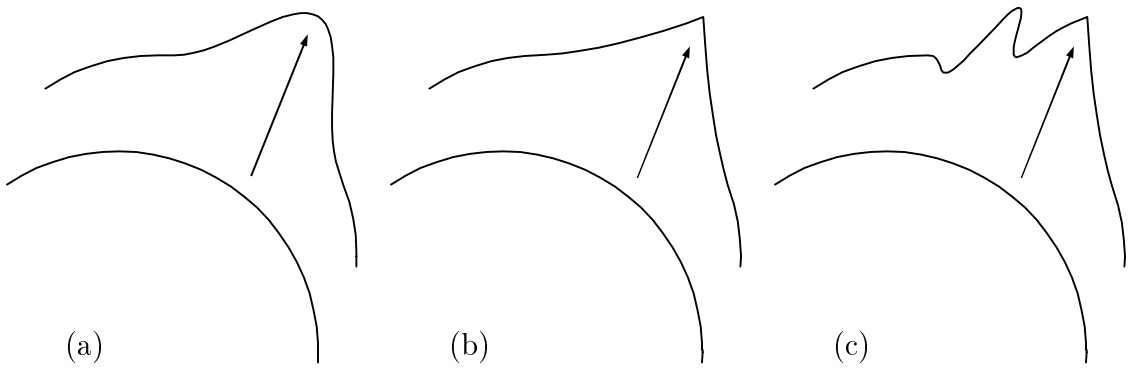


Figure 6.5.1: template drag effects

6.5.1 Template Approximation

To keep the computations tractable for arbitrary numbers of template operations, and to facilitate further design operations after a template is applied, each template's effects should be accumulated in a dyadic-spline approximation. The top-down approximation algorithm of the last chapter is applied to solve this problem. In order to apply the top-down approximation algorithm, local estimates are needed. Examining the basic template formulation

$$\hat{f}(t) = f(t) + cG(t)$$

the approximation process can be reduced to applying the top-down approximation algorithm for the generalized basis function $G(t) = g(h^{-1}(t))$ to obtain the approximation $\tilde{G}(t)$. The output approximation to the basic template formulation then becomes $f(t) + c\tilde{G}(t)$.

If h were in the simple form $h(s) = h_1 s + h_0$ with $h_1 \neq 0$, then $h^{-1}(t) = (t - h_0)/s$. If h is *approximately* in this form over $[s_0, s_1]$, then it could be expressed as $h(s) = h_1 s + h_0 + \delta$ for some interval $\delta = [-\delta_{\max}, \delta_{\max}]$. Then then approximate form of h^{-1} could be expressed in interval form as

$$h^{-1}(t) = (t - h_0 - \delta)/h_1 = (t - h_0)/h_1 + \epsilon$$

where ϵ is the interval

$$\epsilon = [-\delta_{\max}/|h_1|, \delta_{\max}/|h_1|]$$

This first-order approximation of $h^{-1}(t)$ holds for

$$t \in [t_0, t_1] = \begin{cases} [h_1 s_0 + h_0 + \delta_{\max}, h_1 s_1 + h_0 - \delta_{\max}] & \text{if } h_1 > 0 \\ [h_1 s_1 + h_0 + \delta_{\max}, h_1 s_0 + h_0 - \delta_{\max}] & \text{if } h_1 < 0 \end{cases}$$

Now assuming $g(s) = g_1 s + g_0 + \gamma$ holds over $s \in [s_0, s_1]$ for some interval $\gamma = [-\gamma_{\max}, \gamma_{\max}]$, then $g(h^{-1}(\cdot))$ has the first-order approximation

$$\begin{aligned} g(h^{-1}(t)) &= g_1((t - h_0)/h_1 + \epsilon) + g_0 + \gamma \\ &= g_1 t + (g_0 - g_1 h_0/h_1) + (g_1 \epsilon + \gamma) \end{aligned}$$

over the interval $t \in [t_0, t_1]$. This local estimate for $g(h^{-1}(\cdot))$ is readily used for the top-down approximation algorithm.

6.5.2 Offset-Frame Templates

One option to the basic template edit is to allow the template to be applied in a continuously-varying offset frame in place of the constant coordinate frame. This is analogous to offset-frame displacements, but in a continuous rather than discrete formulation. The formula becomes

$$\hat{f}(t) = f(t) + A^{-1}(t_m) A(t) c G(t)$$

where $A(t)$ is a coordinate transform frame that is derived from a smoothed version of the curve $f(t)$, and t_m is the drag point's domain position (e.g. the domain point where $G(t)$ takes on its maximum value). The transform $A^{-1}(t_m)$ is optional, but has the desirable effect that pulling the control vector in the (x, y) plane causes the point $\hat{f}(t)$ to move in the same direction, as would happen when pulling the control vectors of conventional basis functions.

Instead of directly defining $A(t)$ from the normalized tangent of the curve, it is more flexible to allow two stages of smoothing in the definition. The first stage of smoothing allows rough surfaces to be offset in a sensible way, while the second stage fixes the loss of one order of continuity that arises in the usual offsetting process. In the first smoothing stage, the curve is smoothed before obtaining a normalized tangent function. In the second stage, this normalized tangent function is approximated as a dyadic spline in order to allow a second stage of smoothing. The smoothed version of the normalized tangent is then normalized again to obtain the final frame function $A(t)$. These notions are formalized in this section.

Let $f(t)$ represent a parametric curve, and let $\bar{f}(t)$ be the smoothed version of the curve for some smoothing parameter ℓ_{s1}. The normalized tangent to the smoothed

curve is

$$\tau(t) = \frac{\bar{f}'(t)}{\left\|\bar{f}'(t)\right\|}$$

Now the top-down approximation algorithm may be used to obtain $T(t)$ from $\tau(t)$ using interval techniques to compute local estimates. Let $\bar{T}(t)$ be the smoothed version of $T(t)$ for a second smoothing parameter ℓ_{s2}. The continuous, orthonormal offset frame is then defined as

$$A(t) = [u(t)\ v(t)]$$

where $u(t)$ is the unit tangent for the frame:

$$u(t) = \frac{\bar{T}(t)}{\left\|\bar{T}(t)\right\|}$$

The vector $v(t)$ is the unit normal to the smoothed curve at t that is obtained by rotating $u(t)$ counterclockwise by 90 degrees.

The offset template definition adds the template shape to the original curve at an arbitrary segment, and orients this added shape according to the smooth underlying curve. Again, interval techniques allow the result of template offsetting to be approximated using the top-down algorithm. An example is illustrated in Figure 6.5.2. Part (a) shows the template curve $g(s)$ (the domain position map $h(s)$ is a simple linear function). Part (b) shows the smooth underlying curve $\bar{f}(t)$ and offset frames $A(t)$. Part (c) shows the effect of the template offset edit on the original curve.

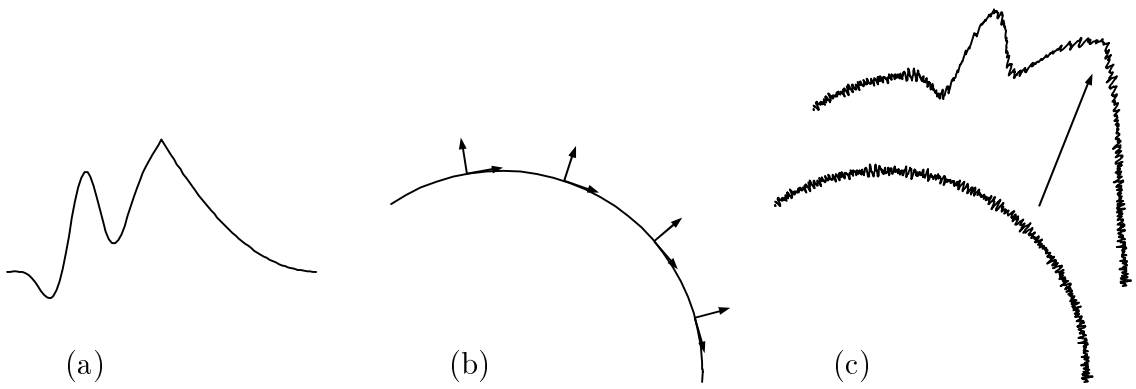


Figure 6.5.2: template offset

6.5.3 Orienting the Details

A second option allows detail orientation to track changes in the smooth underlying curve as the template edit is applied. The idea is to keep details of $f(t)$ that would be removed by smoothing in the simple offset-frame displacement formulation from section 6.3. The template is applied to the smoothed curve (with or without the first template-edit option), where the displacements have been transformed from the simple offset-frame form to standard form. The result is transformed back into the simple offset-frame form, and the detail displacements are restored by simple addition.

The specific approach described in this section preserves details below a smoothing level ℓ_s by separating them out in the simple offset-frame displacement form before applying the template edit operation. After the template edits are performed, the resulting function is placed back into the simple offset-frame displacement format and the details are added back. Figure 6.5.3 illustrates the effect of separating and restoring details in the simple offset-frame displacement format. Part (a) shows the original curve. Part (b) shows the curve with the detail removed. Part (c) shows the effect of a template edit. Finally, part (d) shows the restored detail.

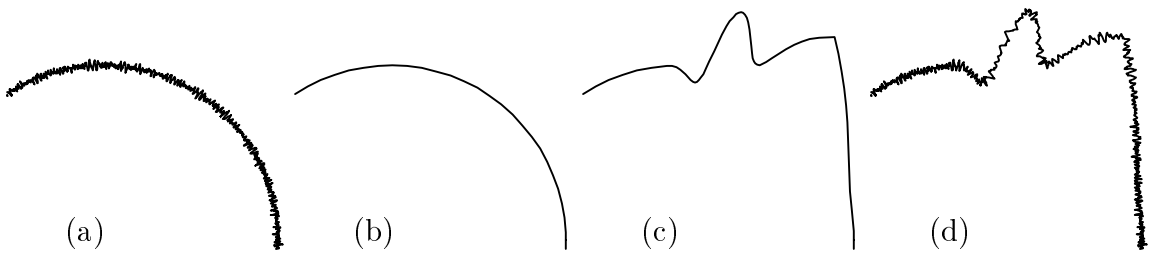


Figure 6.5.3: steps in orienting the details

6.5.4 Combinations of the Optional Template Effects

There are four combinations of the two options described in the previous sections:

1. The base template operation with no options simulates a generalized basis function. The template is applied with a constant orientation, and detail orientation does not track the underlying shape.

2. The first option alone is most closely related to the pasting method described in Barghiel's thesis. The template is applied as a kind of generalized offset, and detail orientation does not track the underlying shape.

3. The second option by itself is most similar to the effect of moving a single displacement in the simple offset-frame formulation, but with a general position, scale and shape of the effect. In this case, detail tracks the orientation of the underlying curve.

4. The first and second options, when combined, give the continuous offset-frame effect and result in details that track the orientation of the underlying curve.

Any one of these combinations might be most desirable in a given application. For comparison, Figure 6.5.4 shows results of the same template edit for the combinations 1–4.

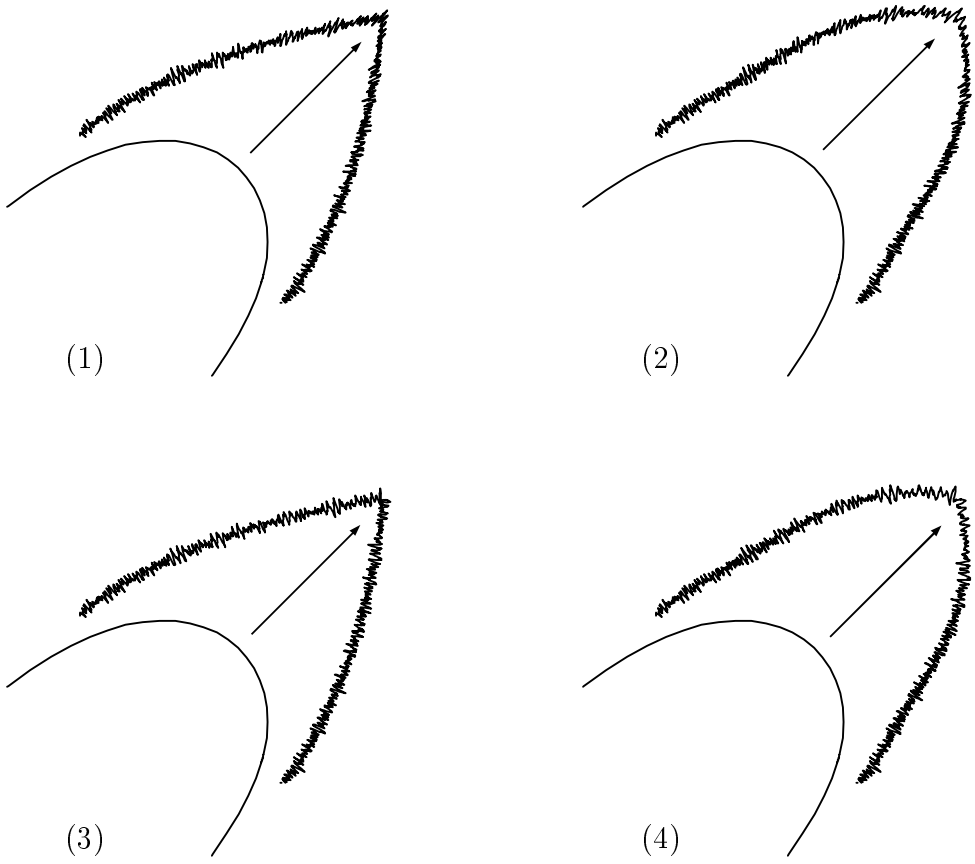


Figure 6.5.4: the four template-edit option combinations

6.6 Sculpting

The last section introduced the general notion of applying a template at arbitrary positions and scales. The approximation method allowed unlimited numbers of these

operations to be applied iteratively without performance penalty and with good accuracy. This suggests that various types of *sculpting* would be feasible, whereby a tool shape modifies a curve repeatedly to emulate various physical effects. Three applications or variants of the template process will be used in this way.

The simplest type of sculpting is to repeatedly push and pull the curve using template offsets, as depicted in the last section in Figure 6.5.1. This is analogous to physical pushing and pulling with a finger (in the case of a smooth interpolating template), or with a sharp object (in the case of the "corner" template). A user can choose the size and shape of the object to apply, and then push and pull one segment at a time.

The simple push-pull approach leaves the detail in a neighborhood intact. Another useful manipulation would smooth out the fine detail as the object is applied. This is analogous to a hammer striking a rough sheet of metal and leaving a smooth impression of the hammer head. This can be accomplished by performing local smoothing on the segment before the "hammer head" template is applied. This effect is shown in Figure 6.6.1.

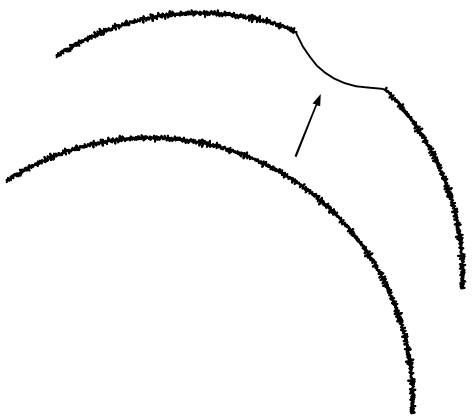


Figure 6.6.1: "hammered" curve

The final type of sculpting operation is analogous to scraping or routing to remove material. A single "scrape" is defined by specifying tool depth in an offset-frame normal direction for each t, where depth zero occurs at a smoothed version of the curve. The offset frame tangent and normal directions $u(t)$ and $v(t)$ are obtained from $A(t)$ in the previous section. The result of scraping is defined by the maximum of the tool depth and the depth of the original curve with respect to the smooth curve.

Let curve $f(t)$ be given and $\bar{f}(t)$ be the smoothed version of f for some smoothing parameter ℓ_s. Let $D_T(t)$ be the given tool depth function, and define the curve depth

as

$$D_C(t) = -v(t) \cdot (f(t) - \bar{f}(t))$$

The result depth will be

$$D(t) = \max\{D_T(t), D_C(t)\}$$

Since the curve position $f(t)$ does not generally reside on the line through $\bar{f}(t)$ in the normal direction $v(t)$, some means of blending from the curve to the scrape ends is needed. A scrape end occurs when $D_T(t) = D_C(t)$. A simple blending method is to linearly move the curve towards the normal line as $D_C(t) - D_T(t)$ goes from positive to zero. The blend factor is defined as

$$q = \begin{cases} 0 & \text{if } D_C(t) - D_T(t) < 0 \\ \frac{D_C(t) - D_T(t)}{H} & \text{if } 0 \le D_C(t) - D_T(t) < H \\ 1 & \text{if } H \le D_C(t) - D_T(t) \end{cases}$$

where H is a user-supplied blend distance. The blend factor is applied to define the scrape result as

$$\hat{f}(t) = \bar{f}(t) + D(t)v(t) + q((f(t) - \bar{f}(t)) \cdot u(t))u(t)$$

Interval estimates are used so that the top-down approximation algorithm may capture the scrape result as a dyadic spline. An example of a single scrape is shown in Figure 6.6.2.

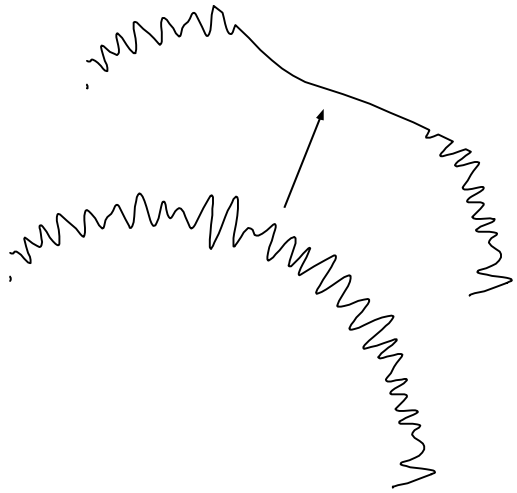


Figure 6.6.2: single "scrape" of tool

To superimpose multiple scrapes as a simultaneous operation (i.e. without altering the smooth underlying surface after each scrape), the tool depth function is taken to be the maximum of the individual scrape tool depth functions

$$D_T(t) = \max_i D_i(t)$$

Otherwise the formulation above remains intact. The result of two simultaneous scrapes is shown in Figure 6.6.3.

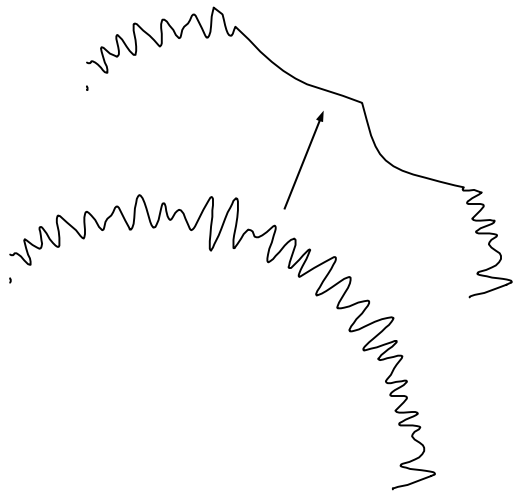


Figure 6.6.3: two simultaneous scrapes

6.7 Summary of Curve Design Results

The editing mechanisms described in this chapter give a great variety of possibilities for curve design. The dyadic-spline formulation combines features of hierarchical B-splines, wavelets and fractals to provide a combination of design effects that would be difficult to achieve with any one of these schemes. Of particular value is the top-down approximation algorithm, which enables several advanced editing effects, and does so with unprecedented efficiency. The dyadic-spline framework provides a unifying methodology for describing this variety of design possibilities. This is important because the unified view provides a language in which new design possibilities, beyond those mentioned in this chapter, can be expressed in simple and effective terms.

Chapter 7

Multivariate Functions

This chapter describes the extension of dyadic splines to multiple variables. A tensor-product construction [15] is used to provide dyadic splines with m-dimensional domains $\Re^m$. The synthesis, analysis and approximation techniques developed for the univariate dyadic splines are straightforward to extend to the tensor-product case. The bintree domain decomposition is ideal for separating the multivariate constructions into a sequence of univariate constructions of the type discussed earlier. The extension of the curve design mechanisms to surfaces is based on the extensions for synthesis, analysis and approximation. Applications are described for fractal construction of natural phenomena.

Tensor-product multivariate functions are formed as piecewise products of univariate functions, with one univariate factor per parameter axis. Section 2.2.6 gave general background information on tensor-product B-splines. For tensor-product B-splines, it is simple and efficient to compute samples, bounds, refinements and the like by treating one axis at a time. For example, for two variables each row of the matrix of control points can be treated as a univariate B-spline for purposes of computation. The columns that result can then be treated as univariate B-splines to compute the final result for two variables. In the case of uniform B-splines, this simple process of separation into univariate computations is organized ideally by m-dimensional bintrees. The separation process is simple to apply to the synthesis, analysis and approximation computations.

7.1 Synthesis

Whereas a univariate bintree decomposition $I_{\ell,i}$ was indexed by level ℓ and index i, the multivariate bintree requires an additional axis counter $a \in \{1, \ldots, m\}$ and multiple indices $i_1, \ldots, i_m$. Recall from section 2.1.1 that the multivariate bintree intervals are

$$I_{\ell,a,i_1,\ldots,i_n} = I_{\ell+1,i_1} \times \cdots \times I_{\ell+1,i_{a-1}} \times I_{\ell,i_a} \times \cdots \times I_{\ell,i_n}$$

To simplify the appearance of this, a shorthand of

$$I_{\mathcal{L},\mathbf{i}} = I_{\ell,a,i_1,\ldots,i_n}$$

will be used, where $\mathcal{L} = (\ell, a)$ and $\mathbf{i} = (i_1, \ldots, i_m)$. The composition $\mathcal{L} = (\ell, a)$ will be refered to as a *layer*, and is analogous to the level in the univariate case. Note that the intervals $I_{\mathcal{L},\mathbf{i}}$ still form a binary tree.

The displacements are now denoted $D_{\mathcal{L},\mathbf{i}}$, and the positions $P_{\mathcal{L},\mathbf{i}}$. Let degrees $n_1, \ldots, n_m$ be specified for each domain axis. The multivariate recurrence is now written as

$$P_{\mathcal{L}\oplus 1} = \mathbf{M}^{\langle a \rangle} P_{\mathcal{L}} + D_{\mathcal{L}\oplus 1}$$

The operator $\mathbf{M}^{\langle a \rangle}$ is made up of copies of the univariate refinement operator for degree n_a uniform B-splines, $\mathbf{M}^{(n_a)}$, as given in section 3.1. One copy of the univariate operator is made for each univariate slice of $P_{\mathcal{L}}$ along axis a. The notation $\mathcal{L} \oplus 1$ refers to the next layer of the bintree

$$\mathcal{L} \oplus 1 = \begin{cases} (\ell, a+1) & \text{if } a < m \\ (\ell+1, 1) & \text{if } a = m \end{cases}$$

Similar notation is used for the previous layer of the bintree:

$$\mathcal{L} \ominus 1 = \begin{cases} (\ell, a-1) & \text{if } a > 1 \\ (\ell-1, m) & \text{if } a = 1 \end{cases}$$

Evaluation of samples and bounds remains the same using interval queries on the bintree intervals, as discussed in sections 3.2 and 3.3. The same notion applies for sampling and bounding: when an interval is not influenced by displacements at lower layers, it can be treated as a local B-spline. Here the neighborhoods of influence of a displacement are formed from cross products of univariate neighborhoods of influence. The interval-query application interface remains the same. Interval-query evaluation also remains the same, based on maintaining families of neighborhoods of influence.

An example bivariate surface is depicted in Figure 7.1.1. The base surface is defined to be planar using the displacements at layer $\mathcal{L} = (0, 1)$. Three additional displacements are nonzero. The local B-spline pieces resulting from the procedure are outlined.

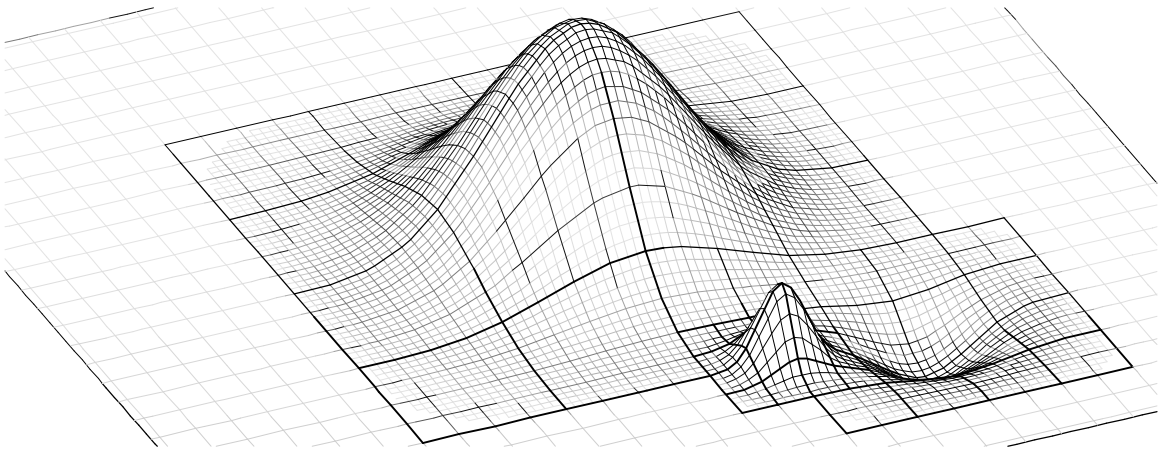


Figure 7.1.1: bivariate dyadic spline and local B-splines

7.2 Analysis

A similar change in notation to that given in the last section provides the definitions for multiresolution analysis for tensor-product dyadic splines. Let the univariate fit operators $\mathbf{F}^{(n_a,k_a)}$ be determined by the axial degrees and filter-width parameters n_a and k_a for $a = 1, \ldots, m$. The multivariate fit operator for axis a, $\mathbf{F}^{\langle a \rangle}$, consists of $\mathbf{F}^{(n_a,k_a)}$ replicated for each univariate slice along axis a. Similar definitions are given for the compaction and expansion operators $\mathbf{C}^{\langle a \rangle}$ and $\mathbf{E}^{\langle a \rangle}$. The compacted displacements will be denoted $Q_{\mathcal{L},\mathbf{i}}$.

The multivariate decomposition step can now be written as

$$\begin{bmatrix} P_{\mathcal{L}} \\ Q_{\mathcal{L}} \end{bmatrix} = \begin{bmatrix} \mathbf{F}^{\langle a \rangle} \\ \mathbf{C}^{\langle a \rangle} \end{bmatrix} P_{\mathcal{L}\oplus 1}$$

The reconstruction step is

$$P_{\mathcal{L}\oplus 1} = \left[\mathbf{M}^{\langle a \rangle} | \mathbf{E}^{\langle a \rangle}\right] \begin{bmatrix} P_{\mathcal{L}} \\ Q_{\mathcal{L}} \end{bmatrix}$$

Note that these definitions immediately satisfy the "reversibility" requirement of a multiresolution analysis filter bank:

$$\begin{bmatrix} \mathbf{F}^{\langle a \rangle} \\ \mathbf{C}^{\langle a \rangle} \end{bmatrix} \left[\mathbf{M}^{\langle a \rangle} | \mathbf{E}^{\langle a \rangle}\right] = \begin{bmatrix} \mathbf{I} & 0 \\ 0 & \mathbf{I} \end{bmatrix}$$

The notions of canonical displacements, displacement fitting and incremental fitting of section 4.4 all carry over as well.

An example bicubic wavelet is depicted in Figure 7.2.1.

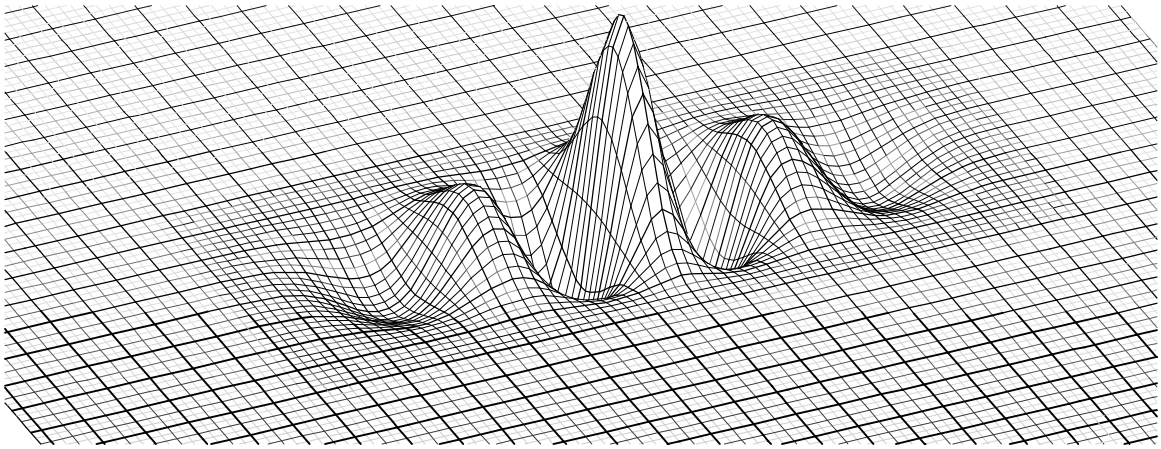


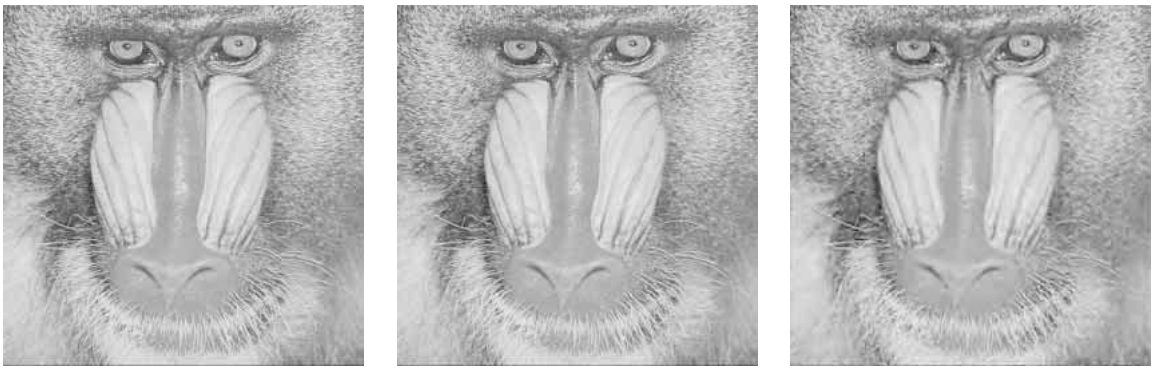
Figure 7.2.1: tensor-product bicubic wavelet

7.3 Approximation

The expected-error bottom-up approximation algorithm of section 5.1 works without modification for tensor-product dyadic splines. In Figure 7.3.1 the full mandrill image $f(u, v)$ is approximated for biquadratic dyadic splines with filter-width parameter $k = 2$ on each axis. The least-squares error norm $e = \left\| f - \hat{f} \right\|_2$ and number of nonzero compacted displacements N is indicated for the progression of approximations. A plot of accuracy versus number of compacted displacements is shown in Figure 7.3.2. As in the univariate case, accuracy is measured as the mean-square signal-to-noise ratio in decibels

$$10 \log_{10}(\|f - f_a\|_2^2 / \left\| \hat{f} - f \right\|_2^2)$$

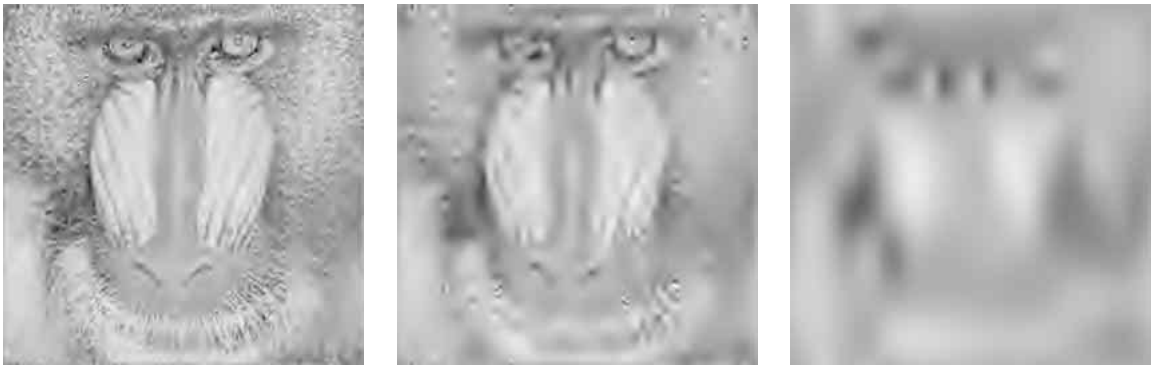
where f_a is the average value f takes on.



original
$N = 262143$

$N = 52461$
$e = 20.79$

$N = 10468$
$e = 40.68$



$N = 2058$
$e = 51.50$

$N = 374$
$e = 57.40$

$N = 41$
$e = 68.19$

Figure 7.3.1: mandrill image approximation sequence

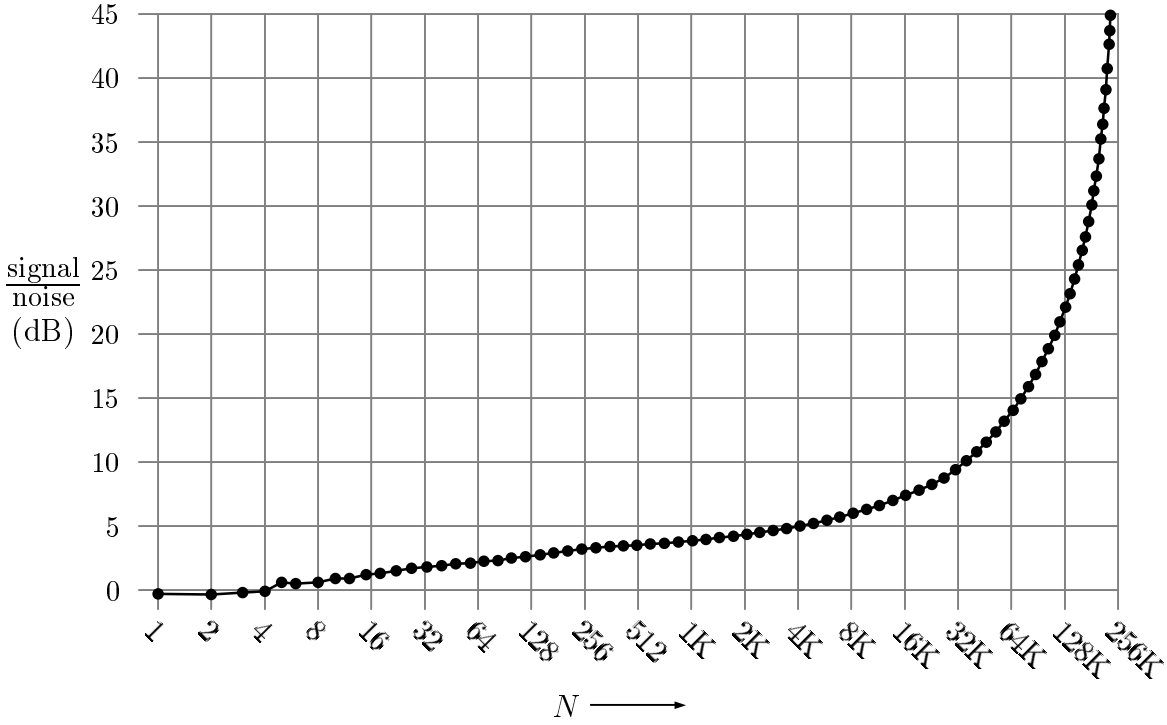


Figure 7.3.2: signal/noise ratio (dB) versus N

The tensor-product extension of the top-down approximation algorithm of section 5.2 is based on tensor-product extensions to the local estimates and limit-fit operators. The top-down algorithm requires that local estimates be available for the target function as before, but now in tensor-product Bernstein/Bézier form [15]. The process of limit fitting on these estimates is extended to the tensor-product setting by obtaining β fit weights for each axial direction and applying these along each set of axis slices in turn. This is exactly analogous to the application of the tensor-product fit operator, but requires that the estimates also be dealt with by separating their basis functions into axial factors. The estimate-fit kernel computation can then be separated into the univariate computations of section 5.2.2. The phases of the top-down approximation algorithm remain the same.

An example of top-down fitting is shown in Figure 7.3.3. The target function is made up of two superimposed cones. Top-down approximation is used to define several surface design operations, as discussed in the next section.

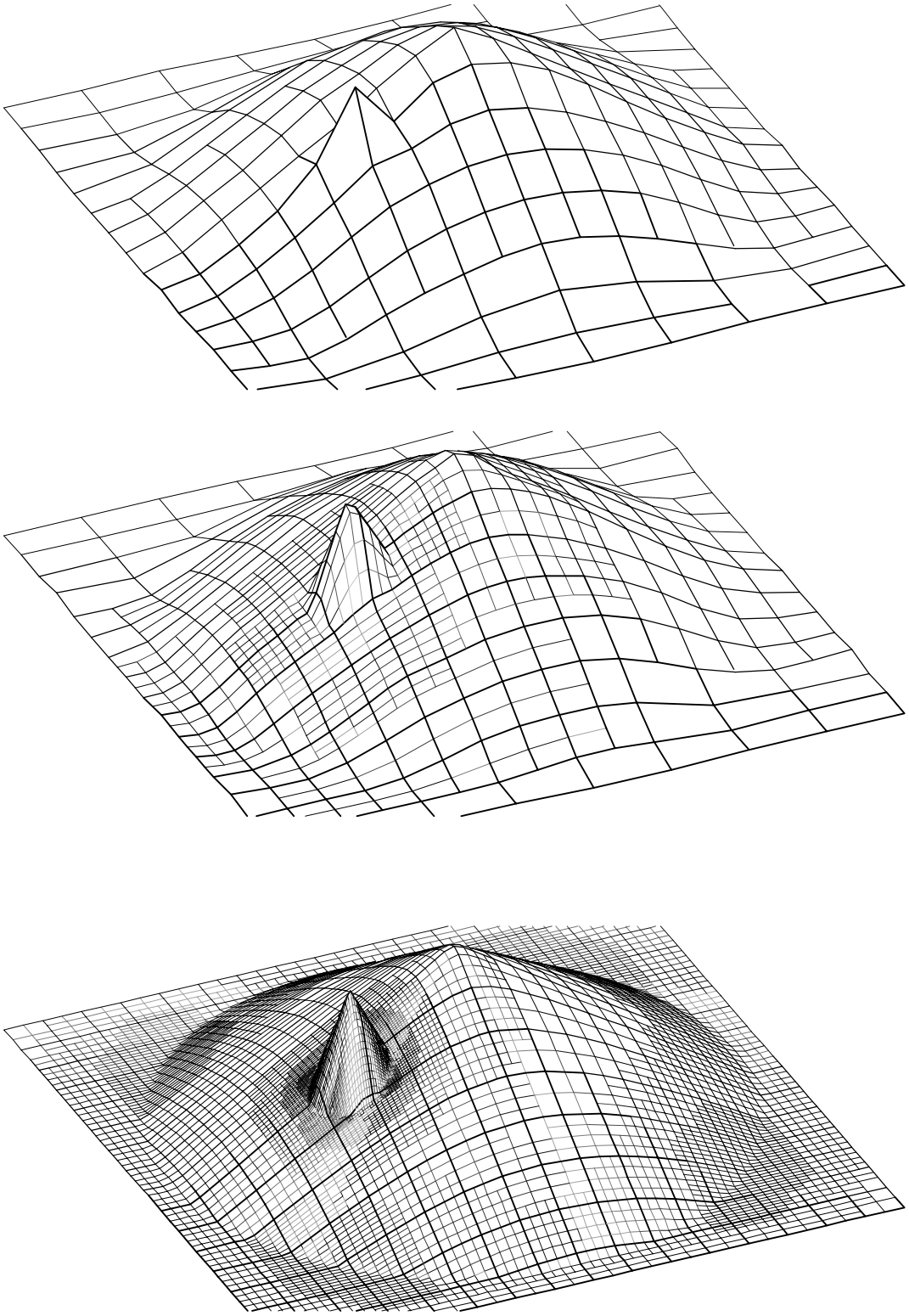


Figure 7.3.3: top-down surface approximation

7.4 Surface Design

Each of the design mechanisms discussed in Chapter 6 has a natural extension to tensor-product dyadic-spline surfaces. Figure 7.4.1 illustrates the six mechanisms: displacement edits, local smoothing, local roughening, offset-frame displacement edits, template edits and sculpting.

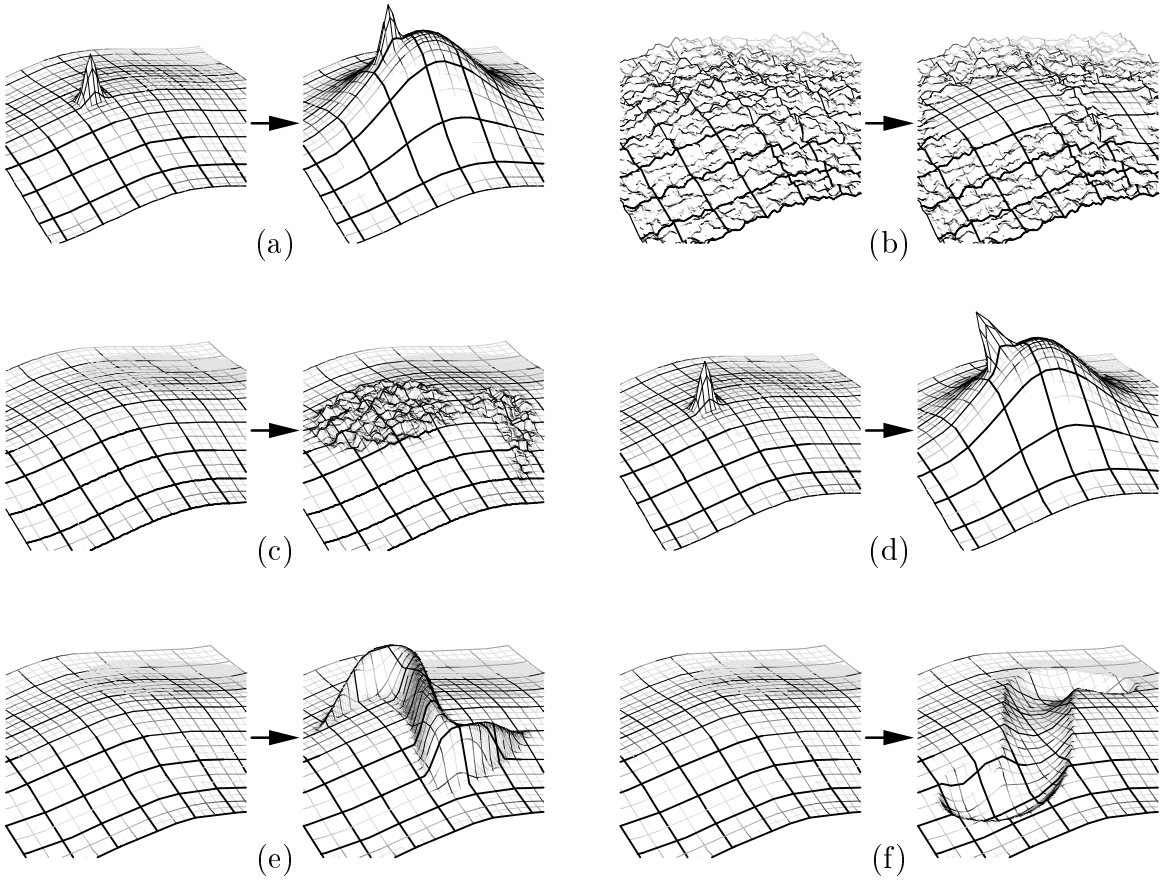


Figure 7.4.1: surface design mechanisms

For this section let the surface be parameterized over u and v, and take on values in three space with coordinates x, y, z:

$$f(u,v) = \begin{bmatrix} f_x(u,v) \\ f_y(u,v) \\ f_z(u,v) \end{bmatrix}$$

Let the partial derivatives with respect to u and v be denoted f_u and f_v respectively. Let u correspond to $a = 1$ and v correspond to $a = 2$.

7.4.1 Displacement Edits

Direct edits of surface displacements follow the same development as the displacements for curves in section 6.1. An example of a displacement edit is shown in Figure 7.4.2.

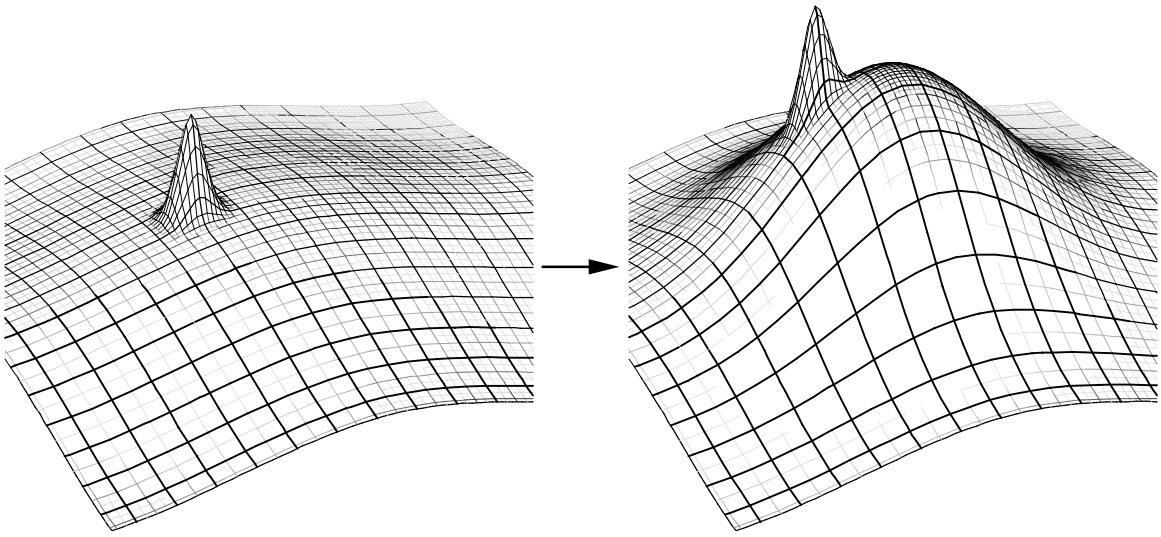


Figure 7.4.2: displacement edit

7.4.2 Smoothing

Global smoothing of surface displacements is similar to the curve case described in section 6.2. The primary difference is the treatment of layer $\mathcal{L} = (\ell, a)$ compared to curve level ℓ. Note that there are two axes ($a = 1$ and $a = 2$). In order to avoid axis order-dependent artifacts, layers $(\ell, 1)$ and $(\ell, 2)$ will be treated the same for purposes of smoothing. Thus global smoothing is defined for smoothing parameter ℓ_s as

$$\bar{D}_{\mathcal{L},\mathbf{i}} = \begin{cases} D_{\mathcal{L},\mathbf{i}} & \text{if } \ell < \ell_s \\ (\ell_s - (\ell - 1))D_{\mathcal{L},\mathbf{i}} & \text{if } \ell - 1 < \ell_s \leq \ell \\ 0 & \text{otherwise } (\ell_s \leq \ell - 1) \end{cases}$$

For local smoothing, a generalization of the smoothing segment is needed. For this, a smoothing area is defined using the concept of a trim curve [4], previously used in the methods for trimmed surface patches. A trim curve $c(t)$ is a continuous, periodic mapping from $t \in [0, 1]$ to the surface domain $(u, v) \in \Re^2$. This curve encloses a domain area that will serve as the locality to be smoothed.

The smoothing operation blends between the original displacements $D_{\mathcal{L},\mathbf{i}}$ and the smoothed ones $\bar{D}_{\mathcal{L},\mathbf{i}}$. The blend factor q is defined as the fraction of $D_{\mathcal{L},\mathbf{i}}$'s interval of influence I_D that overlaps the area enclosed by $c(t)$. The computation of this overlap is discussed at the end of this section. The blend factor q is then applied as in the curve case:

$$\breve{D}_{\mathcal{L},\mathbf{i}} = (1-q)D_{\mathcal{L},\mathbf{i}} + q\bar{D}_{\mathcal{L},\mathbf{i}}$$

Some results of local smoothing are shown in Figure 7.4.3.

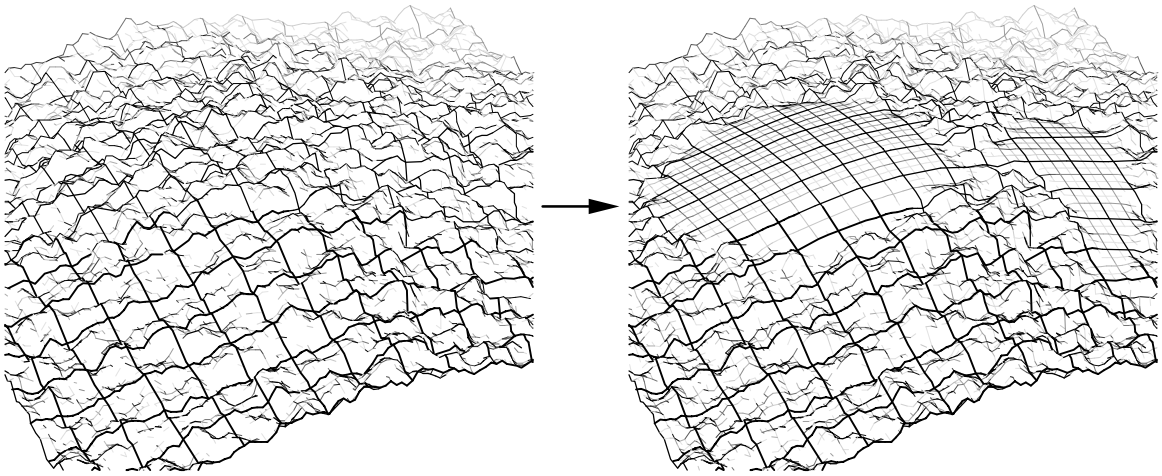


Figure 7.4.3: local surface smoothing

Computing the Blend Factor

It is nontrivial to compute the area of overlap of an interval I and the area enclosed by a trim curve $c(t)$. However, the well-known Warnock algorithm for polygon visibility [37] can be adapted to this problem. Although the concern here is only for determining the area of a single "polygon" $c(t)$ within a "view window" I, the Warnock algorithm has a useful property of dividing I into smaller intervals until each interval either misses $c(t)$, $c(t)$ crosses the interval in a simple way, or the interval is small. Winding number computations are used in this algorithm to determine which intervals (or which parts of crossed intervals) are inside the trim curve. To apply the polygon techniques to a curve, the curve must be approximated by a polygon. For the purposes of interactive editing, it is sufficient to ensure that the approximation error is within a small fraction of the width of the interval I. If $c(t)$ is a dyadic spline, or is in B-spline form, standard subdivision techniques can be applied to accomplish this [15]. In the implementation used here, "simple" crossings consist of two or fewer polygon edges, and a bintree decomposition of I is used. An example Warnock-style decomposition of a trim area is shown in Figure 7.4.4.

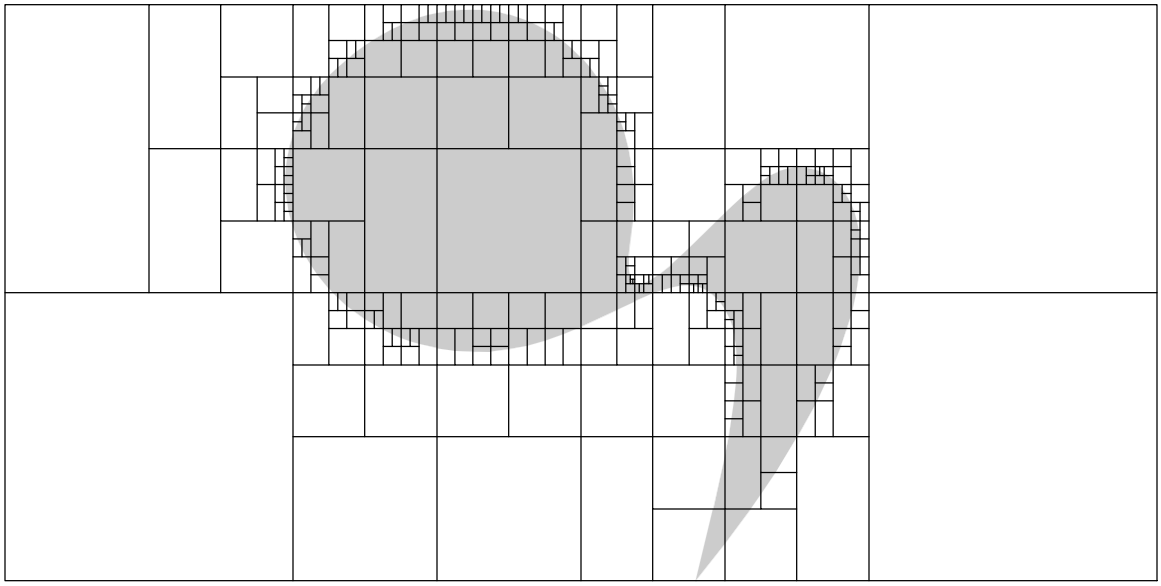


Figure 7.4.4: Warnock decomposition of trim area

7.4.3 Offset-Frame Displacements

Recall from section 6.3 that offset frames for curve displacements were defined as the unit tangent and normal at the point of maximum influence for the displacement, taken with respect to a smoothed version of the curve with smoothing parameter ℓ_s. Similarly, the offset frame for a surface displacement $D_{\mathcal{L},\mathbf{i}}$ is defined as the two unit tangents and unit normal at the point of maximum influence, taken with respect to a smoothed version of the surface.

Let $f(u,v)$ be the surface and $\bar{f}(u,v)$ be a smoothed version of the surface for smoothing parameter ℓ_s. Then the offset coordinate frame applied to offset displacement $\hat{D}_{\mathcal{L},\mathbf{i}}$ is then defined as

$$A_{\mathcal{L},\mathbf{i}} = [\mathbf{p}\ \mathbf{q}\ \mathbf{r}]$$

where

$$\begin{aligned}
\mathbf{p} &= \frac{\bar{f}_u(u_m,v_m)}{\left\|\bar{f}_u(u_m,v_m)\right\|} \\
\mathbf{q} &= \frac{\bar{f}_v(u_m,v_m)}{\left\|\bar{f}_v(u_m,v_m)\right\|} \\
\mathbf{r} &= \frac{\mathbf{p}\times\mathbf{q}}{\|\mathbf{p}\times\mathbf{q}\|}
\end{aligned}$$

and (u_m, v_m) is the domain point of maximum influence. Now the application of $A_{\mathcal{L},\mathbf{i}}$ to $\hat{D}_{\mathcal{L},\mathbf{i}}$ gives the standard displacement as

$$D_{\mathcal{L},\mathbf{i}} = A_{\mathcal{L},\mathbf{i}} \hat{D}_{\mathcal{L},\mathbf{i}}$$

This gives the effect that details track the position and orientation of the smooth underlying surface. An edit in the offset-frame representation is shown in Figure 7.4.5.

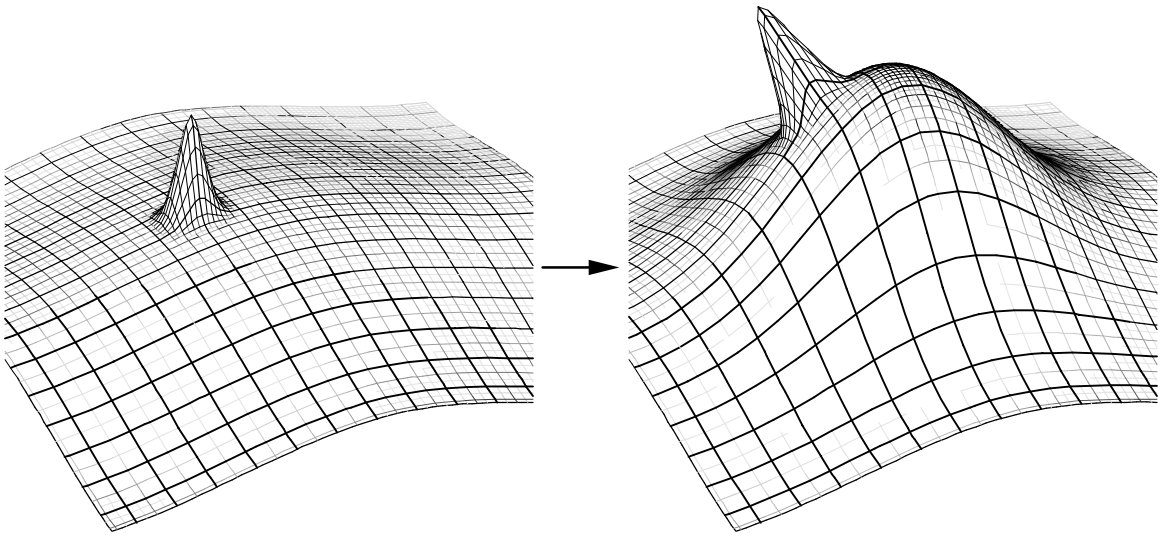


Figure 7.4.5: offset-frame detail tracking

7.4.4 Roughening

The extension of the roughening operation to surfaces is similar to the extension for smoothing. Global roughening is produced in the surface case by replacing detail compacted displacements with fractal compacted displacements that point in the offset normal direction, just as in the curve case. The localization of the roughening effect is accomplished in the same manner as local surface smoothing. An example of local roughening is shown in Figure 7.4.6.

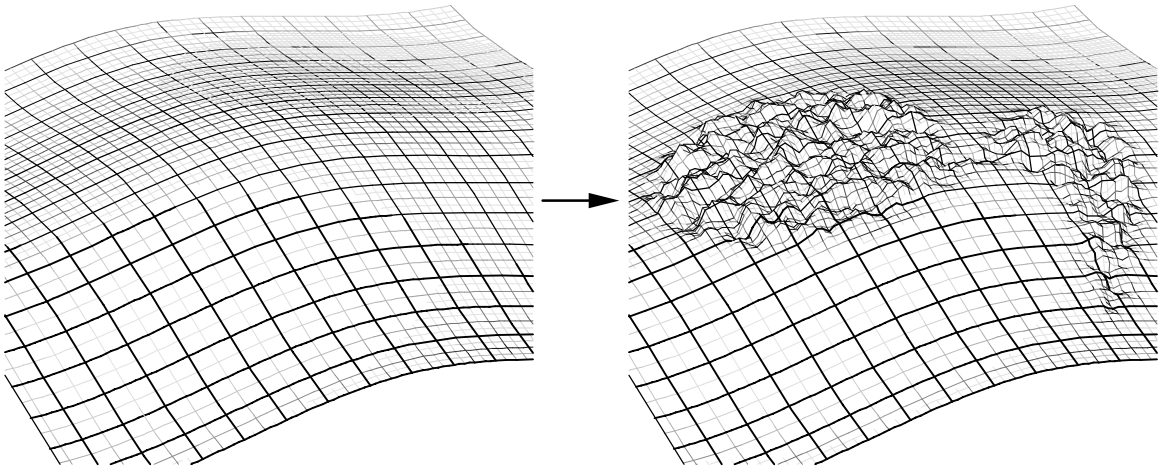


Figure 7.4.6: local surface roughening

7.4.5 Template Edits

Template editing extends naturally to a surface $f(u, v)$. Let $g(s, t)$ be a scalar-valued template function and let $h(s, t)$ be an invertible domain-positioning function. The basic template edit effect is defined as

$$\hat{f}(u, v) = f(u, v) + cG(u, v)$$

where c is a control vector for the *generalized basis function*

$$G(u, v) = g(h^{-1}(u, v))$$

Note that $h^{-1}(u, v)$ is only defined for $(u, v) \in h(J)$ where J is the interval domain of $h(s, t)$. When appropriate, assume that $G(u, v)$ is zero when $(u, v) \notin h(J)$. Also note that the control vector c may be derived from another control vector $\hat{c}$ that is defined in a local offset frame A, similar to the offset-frame displacement edits of section 7.4.3.

The result of a template edit, $\hat{f}(u, v)$, is approximated using the top-down algorithm. As with curve templates in section 6.5.1, local estimates are formed using interval-analytic techniques. This approximation process is described in the next section. An example result of template editing is depicted in Figure 7.4.7.

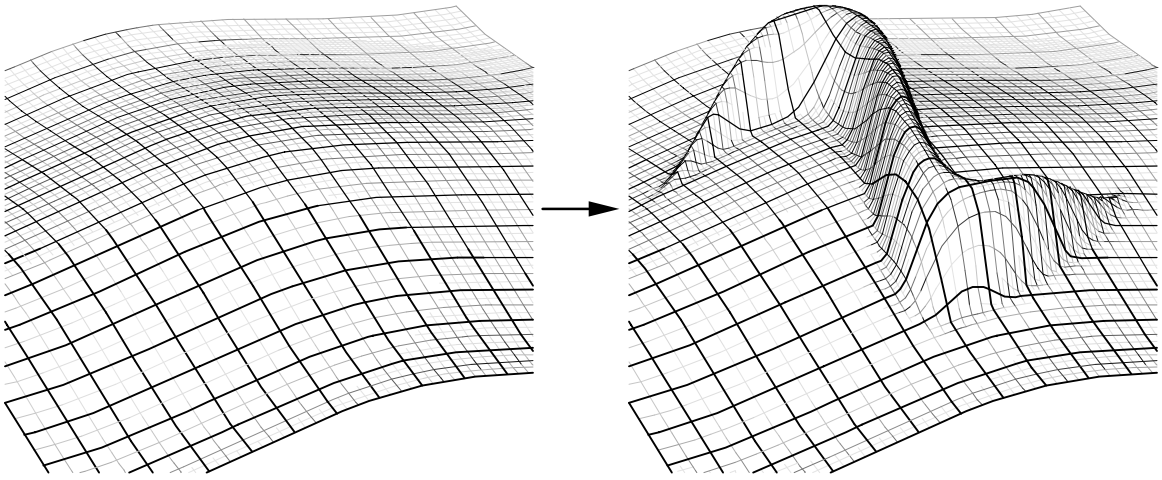


Figure 7.4.7: surface template edit

Template Edit Approximation

The template edit result $\hat{f}(u,v)$ is approximated by using the top-down algorithm to approximate the generalized basis function $G(u,v) = g(h^{-1}(u,v))$. This approximation, denoted $\tilde{G}(u,v)$, scales a control vector c before added it to $f(u,v)$. The approximate template-edit result is $\hat{f}(u,v) = f(u,v) + c\tilde{G}(u,v)$. Applying the top-down algorithm to approximate $G(u,v)$ reduces to finding local estimates. The remainder of this section will discuss the computation of suitable local estimates.

This discussion will use first-order interval estimates throughout. To develop an estimate for $g(h^{-1}(u,v))$, an estimate will first be constructed for $h^{-1}(u,v)$ based on an estimate of $h(s,t)$. This will be composed with an estimate of $g(s,t)$ to give the desired estimate of $g(h^{-1}(u,v))$.

Let $h(s,t)$ have the first-order interval estimate

$$\tilde{h}(s,t) = H \begin{bmatrix} s \\ t \end{bmatrix} + \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} + \delta$$

where H is an invertible 2×2 matrix, and δ is an interval in (u,v) space. Assume that this estimate holds for $h^{-1}(I)$, where I is an interval in (u,v) space. An interval estimate for $h^{-1}(u,v)$ is

$$\tilde{h}^{-1}(u,v) = H^{-1} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} s_0 \\ t_0 \end{bmatrix} + \epsilon$$

where

$$\begin{bmatrix} s_0 \\ t_0 \end{bmatrix} = -H^{-1} \begin{bmatrix} u_0 \\ v_0 \end{bmatrix}$$

and where the error interval ϵ is chosen so that

$$\epsilon \supset -H^{-1}\delta$$

This estimate holds for $(u, v) \in I$. The error ϵ may be computed as the bounding box of the image of the four corners of δ under the transform $-H^{-1}$.

Now suppose $g(s, t)$ has the estimate

$$\tilde{g}(s, t) = [g_s \; g_t] \begin{bmatrix} s \\ t \end{bmatrix} + g_0 + \gamma$$

for error interval γ, and suppose this holds for $(s, t) \in h^{-1}(I)$. Then an estimate for $g(h^{-1}(u, v))$ over I is

$$[g_s \; g_t] \left(H^{-1} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} s_0 \\ t_0 \end{bmatrix} + \epsilon \right) + g_0 + \gamma$$

Offset-Frame Templates

The two options for curve templates also apply to surface templates. Allowing detail orientation to track the coarse underlying surface works exactly as in the curve case. The continuous offset-frame option is also straightforward, as follows.

Smooth, continuous offset frames are defined using two stages of smoothing in forming a normalized tangent, just as in the curve case. The first stage gives reasonable offsets for a rough surface, while the second stage fixes the loss of one order of continuity that results from the usual offsetting process. For a surface $f(u, v)$, let $\bar{f}(u, v)$ be the smoothed version of the surface for smoothing parameter ℓ_{s1}. The tangents of this smoothed surface are normalized to give

$$\hat{\mathbf{p}}(u, v) = \frac{\bar{f}_u(u, v)}{\left\| \bar{f}_u(u, v) \right\|}$$

$$\hat{\mathbf{q}}(u, v) = \frac{\bar{f}_v(u, v)}{\left\| \bar{f}_v(u, v) \right\|}$$

These normalized tangents are approximated as dyadic splines (using the top-down algorithm) to allow the second stage of smoothing. Let $\tilde{\mathbf{p}}(u, v)$ and $\tilde{\mathbf{q}}(u, v)$ be the approximations to the normalized tangents, and $\bar{\mathbf{p}}(u, v)$ and $\bar{\mathbf{q}}(u, v)$ be the smoothed versions of these for smoothing parameter ℓ_{s2}. A final normalization and cross product gives the axis vectors of the desired offset frame

$$A(u, v) = [\mathbf{p}(u, v) \; \mathbf{q}(u, v) \; \mathbf{r}(u, v)]$$

where

$$\begin{aligned}
\mathbf{p}(u,v) &= \frac{\bar{\mathbf{p}}(u,v)}{\|\bar{\mathbf{p}}(u,v)\|} \\
\mathbf{q}(u,v) &= \frac{\bar{\mathbf{q}}(u,v)}{\|\bar{\mathbf{q}}(u,v)\|} \\
\mathbf{r}(u,v) &= \frac{\mathbf{p}(u,v) \times \mathbf{q}(u,v)}{\|\mathbf{p}(u,v) \times \mathbf{q}(u,v)\|}
\end{aligned}$$

The continuous offset-frame template edit becomes

$$\hat{f}(u,v) = f(u,v) + A^{-1}(u_m, v_m)A(u,v)cG(u,v)$$

where (u_m, v_m) is the domain point of maximum influence for $G(u,v)$. As in the curve case, the transform $A^{-1}(u_m, v_m)$ is optional, but has the desirable effect that pulling the control vector c in (x,y,z) space causes the point $\hat{f}(u,v)$ to move in the same direction, as would happen when pulling the control vectors of conventional basis functions.

7.4.6 Sculpting

The three types of sculpting described for curves extend easily to surfaces. The first two, push-pull template edits and "hammering" are exactly the same as for curves. The remainder of this section will discuss the third and more complex sculpting operation that simulates "scraping."

A single surface "scrape" is defined by specifying tool depth in an offset-frame normal direction for each (u,v), where depth zero occurs at a smoothed version of the surface. The offset frame tangent and normal directions $\mathbf{p}(u,v)$, $\mathbf{q}(u,v)$ and $\mathbf{r}(u,v)$ are obtained from $A(u,v)$ in the previous section. The result of scraping is defined by the maximum of the tool depth and the depth of the original surface with respect to the smooth surface.

Let $f(u,v)$ be a given surface and $\bar{f}(u,v)$ be the smoothed surface for some smoothing parameter ℓ_s. Let $D_T(u,v)$ be the given tool depth function, and define the surface depth as

$$D_S(u,v) = -\mathbf{r}(u,v) \cdot (f(u,v) - \bar{f}(u,v))$$

The result depth will be

$$D(u,v) = \max\{D_T(u,v), D_S(u,v)\}$$

Since the surface position $f(u,v)$ does not generally reside on the line through $\bar{f}(u,v)$ in the normal direction $\mathbf{r}(u,v)$, some means of blending from the surface to the scrape

boundary is needed. A scrape boundary occurs when $D_T(u,v) = D_S(u,v)$. A simple blending method is to linearly move the surface towards the normal line as $D_S(u,v) - D_T(u,v)$ goes from positive to zero. The blend factor is defined as

$$q = \begin{cases} 0 & \text{if } D_S(u,v) - D_T(u,v) < 0 \\ \frac{D_S(u,v) - D_T(u,v)}{H} & \text{if } 0 \leq D_S(u,v) - D_T(u,v) < H \\ 1 & \text{if } H \leq D_S(u,v) - D_T(u,v) \end{cases}$$

where H is a user-supplied blend distance. The blend factor is applied to define the scrape result as

$$\begin{aligned} \hat{f}(u,v) \;=\; & \bar{f}(u,v) + D(u,v)\mathbf{r}(u,v) + \\ & q((f(u,v) - \bar{f}(u,v)) \cdot \mathbf{p}(u,v))\mathbf{p}(u,v) + \\ & q((f(u,v) - \bar{f}(u,v)) \cdot \mathbf{q}(u,v))\mathbf{q}(u,v) \end{aligned}$$

Interval estimates are used so that the top-down approximation algorithm may capture the scrape result as a dyadic spline. An example of a single scrape is shown in Figure 7.4.8.

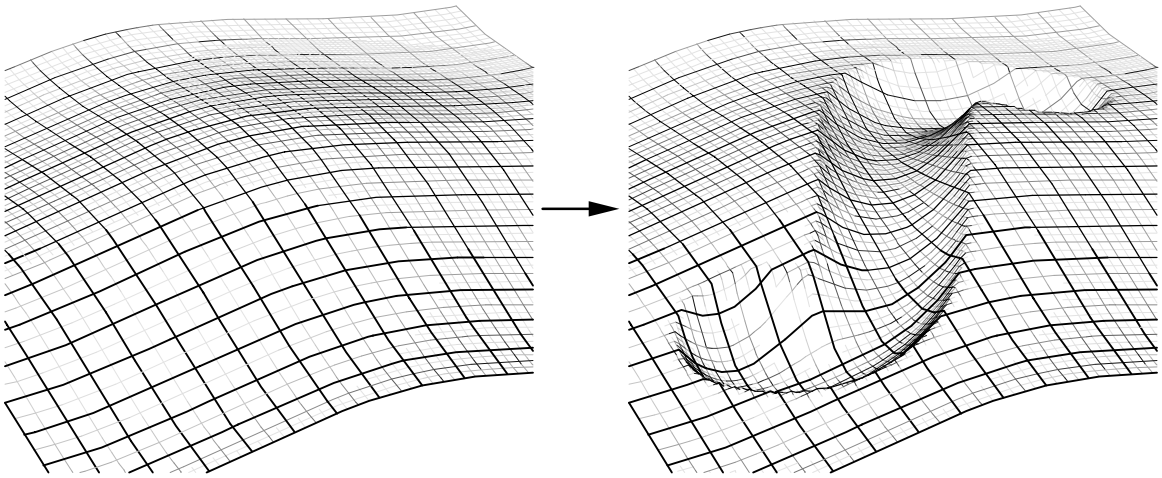


Figure 7.4.8: single surface "scrape"

As with curve scrapes, superimposing multiple scrapes as a simultaneous operation is performed by letting the tool depth function be defined as the maximum of the individual scrape tool depth functions

$$D_T(u,v) = \max_i D_i(u,v)$$

Otherwise the formulation above remains intact. The result of two simultaneous scrapes is shown in Figure 7.4.9.

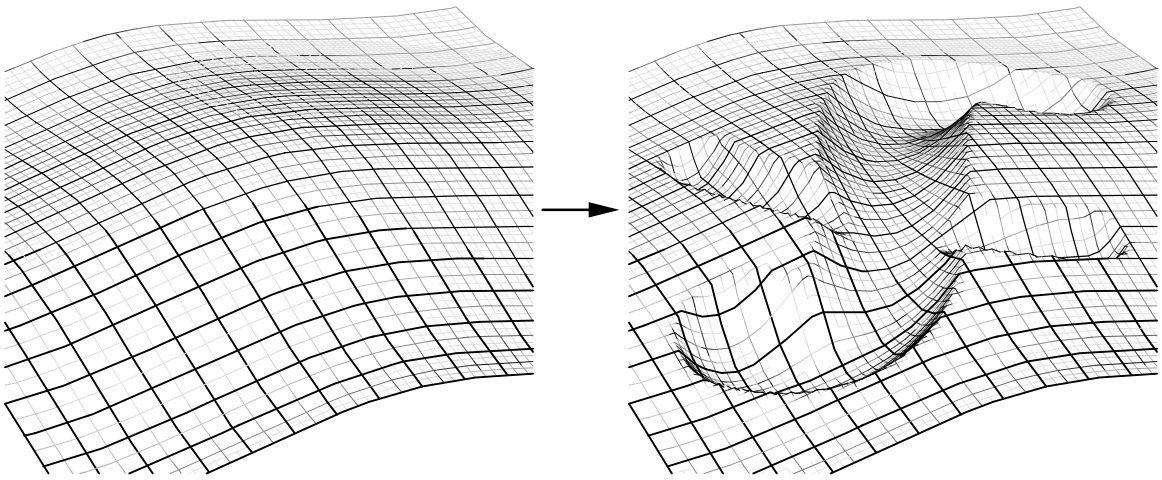


Figure 7.4.9: simultaneous surface scrapes

7.5 Applications of Dyadic-Spline Fractals

Dyadic splines may be used to produce fractals by incorporating automatically-generated random displacements (see [13]). The idea is to choose each random displacement $D_{\ell,i}$ from a uniform distribution over an interval $[-U_\ell, U_\ell]$ that shrinks by half for each successive level of resolution ($U_\ell = \mu 2^{-\ell}$ for a chosen $\mu > 0$). Evaluation for this infinite collection of nonzero displacements is discussed in section 3.2.

The figures in this section demonstrate three distinct uses of the dyadic-spline fractal formulation. Each is distinct in terms of both the phenomena modeled by the fractal and the rendering system that the dyadic-spline implementation is integrated into.

Figure 7.5.1 shows a landscape and cloudy sky synthesized with bivariate fractals. The terrain interprets the function as altitude, while the clouds are based on opacity. The rendering scheme is a form of the painter's algorithm encorporating transparency.

Figure 7.5.2 exhibits two marble textures synthesized by trivariate fractals. Both map the function value into a material color. Rendering is provided by a ray tracer encorporating Monte Carlo integration for antialiasing and soft shadows.

Figure 7.5.3 shows a trivariate fractal interpreted as an implicitly defined solid. The fractal has been hierarchically edited to create the cavernous void. Rendering is accomplished by intersecting rays using interval analysis on neighborhoods of space surrounding ray segments.

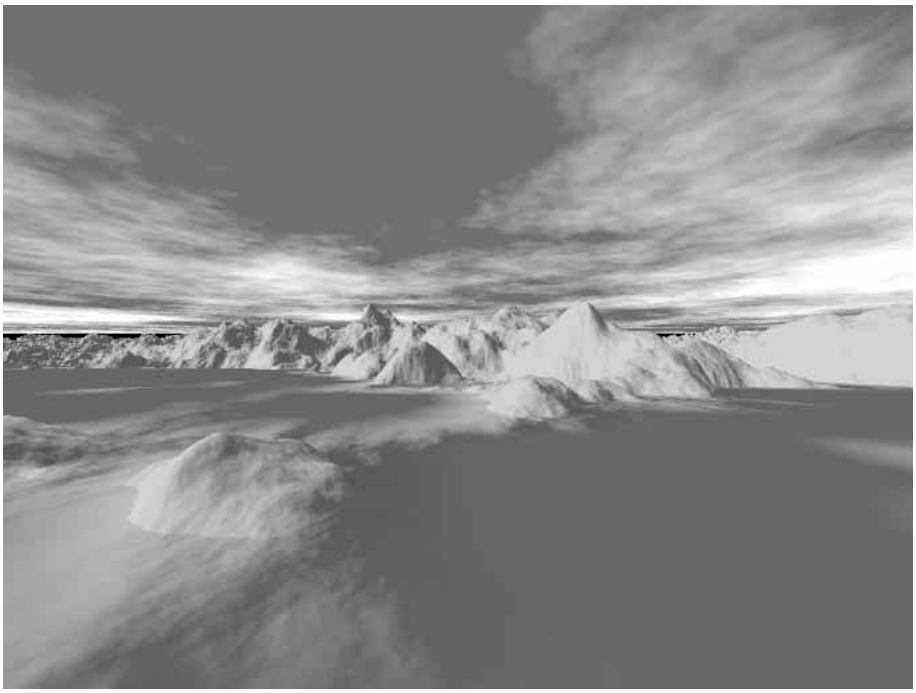


Figure 7.5.1: Skyland

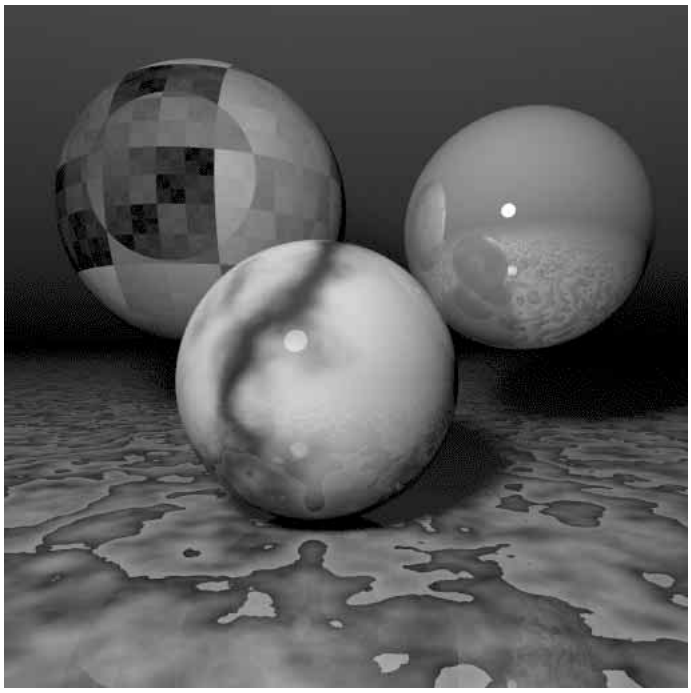


Figure 7.5.2: Marball

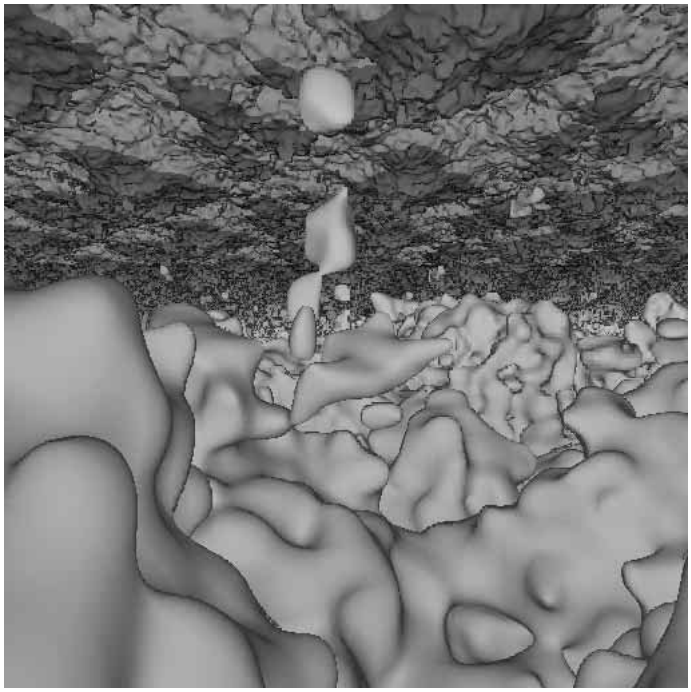


Figure 7.5.3: Cavern

7.6 Summary of Tensor-Product Extensions

This chapter extends the univariate dyadic-spline methods to the tensor-product setting. The discussions of synthesis, analysis and approximation extend naturally to m-variate domains. The bintree domain composition is ideal for organizing these extensions. The curve design mechanisms extend to tensor-product bivariate surfaces. These extensions are straightforward in many cases, such as global smoothing and roughening. In the case of local smoothing and roughening, the powerful Warnock algorithm is adapted to compute local smoothing and roughening effects. This solution performs computations quickly, but is more complex to implement than the simple interval-overlap computation for curves. An increase in implementation complexity also occurs for template edit approximations, particularly in the evaluation of local estimates. Multivariate dyadic splines are useful for modeling natural phenomena as fractals. Because of the interval-query evaluation interface, the dyadic-spline fractals intergrate smoothly into diverse image synthesis systems.

Chapter 8

Conclusions and Future Work

This dissertation has presented the dyadic-spline function representation and a body of algorithms to support its application. Dyadic splines are defined and controlled by displacements that may be specified at any level of resolution. Using uniform B-spline refinement, these displacements are converted into a hierarchy of range positions through a doubling and perturbing process. An efficient and flexible evaluation mechanism, interval queries, provides applications with a wealth of information about the how the function behaves over neighborhoods (corner-point samples, range bounds, etc.).

The question of how to convert known target functions into dyadic splines has been answered in both exact and approximate senses. Exact analysis of a target function is provided by least-squares level-to-level fitting, so that positions at one level of resolution best predict those at the next finer level. A compaction method is introduced that eliminates a factor-of-two redundancy in the displacement representation. The resulting compacted displacements form a type of wavelet representation. An enhancement to the analysis and complimentary synthesis processes allows applications to trade off computation speed versus closeness to an ideal least-squares fitting process. Even with non-ideal fitting, perfect level-to-level predictions are found whenever they exist, and the target function is exactly reproduced.

Two approximation algorithms allow tradeoff of complexity versus accuracy of reproduction of the target function. A simple bottom-up algorithm starts from highly detailed knowledge of the target function, performs the wavelet-style analysis, and sets to zero the smallest-magnitude compacted displacements until the desired accuracy or complexity threshold is reached. A much faster top-down algorithm works by carefully adding compacted displacements until the desired accuracy or complexity threshold is reached. This processing is based on "fuzzy" knowledge of the target functions in the form of bounded local estimates. Compacted displacements are "split" into new ones to add more detail in neighborhoods that are out of tolerance. Old compacted displacements are improved as new local estimates become less "fuzzy." The top-down algorithm runs in time that is linear in the complexity of the output approximation. This can be hundreds or thousands of times faster than the bottom-up approximation, which runs in time that is linear in the complexity of highly detailed knowledge of the target function (typically, most of this detail is thrown away). The approximations' tradeoff of accuracy versus complexity is the basis for lossy compression.

The synthesis, analysis and approximation tools have natural extensions to multiple variable using a tensor-product formulation. The theory and algorithms generalize with little or no change.

The synthesis, analysis and approximation tools are applied to the tasks of curve and surface design. A wide variety of editing mechanisms are introduced that combine and extend features of several earlier function representations. The top-down approximation algorithm is of great importance in providing many of the most powerful design mechanisms, including sculpting and the application of general template shapes to an object.

8.1 Outline of Future Research

Dyadic splines and the associated algorithms have many potential applications beyond those touched on in this dissertation. Compression of many types of data is possible, including medical images, elevation data, physical simulation results and video. Dyadic splines could play a key role in the efficient computation of solutions to difficult physical equations, such as global illumination or fluid dynamics.

In addition to applications research, numerous improvements and extensions are needed for the method itself. The primary improvements needed regard evaluation, generalization and the top-down approximation algorithm.

The interval-query evaluation method is applicable to a diverse collection of applications. It automatically provides information at multiple scales and caches results, as required in many advanced computations. A number of issues have not been addressed in the research to date. The challenges that remain, along with indications of future work, are:

- there is considerable overhead in maintaining the dynamic cache

  The reduction in overhead is an important area for future research. One possibility is to move to coarser-grained data structures, such as blocks of bintree intervals, to gain economy-of-scale efficiencies in both computation time and in storage space.

- there is no way to predict in general if or when reuse will occur

  Improving the prediction of reuse is possible for many applications, as they have considerable regularity in usage. However, automatic estimation of such regularity is difficult. Instead, applications can be given the option to specify which cache entries are likely to be needed in the near future, and this can be used to order the de-allocation queue.

- the memory limit is rarely needed for simple problems

The overhead of the memory-limit mechanism can be reduced by coarser-grained allocation and de-allocation, similar to the efficiencies envisioned to reduce general cache overhead. Also, the memory-limit mechanism overhead can be avoided entirely and automatically for simple problems (only maintain the reuse queue when total memory use exceeds a threshold).

- for local B-spline pieces, direct computation of sparse non-corner samples is more efficient that subdividing until accurate

  The efficiency of sparse random sampling can be improved by performing direct sample computations when appropriate.

The primary generalization envisioned is to extend the dyadic-spline formulation and algorithms to subdivision surfaces, such as those introduced by Catmull and Clark [5]. If an additional generalization were included, allowing sharp features such as the vertex of a cone or the edge of a cylinder, then a single generalized dyadic spline could elegantly capture most of the geometric shapes found in solid modeling and computer graphics. Some initial work along these lines has been performed by researchers at the University of Washington [25, 23, 14].

Four areas of research are clear for the top-down approximation algorithm:

- usefulness for other wavelets

  The top-down approximation algorithm should be applicable to wavelet representations other than the dyadic splines. Only two parts of the top-down algorithm have some sensitivity to the wavelets chosen: the comparison of the wavelet approximation versus the local estimate, and the incremental, sparse updates to the wavelet coefficients as move active wavelets are added during processing. It seems likely that these issues can be solved for many wavelet schemes.

- tuning for various norms

  The choices of which domain intervals to split and which intervals are "done" should be made with the desired norm in mind. This seems to be fairly straightforward, but has not been investigated so far.

- optimization of accuracy/complexity tradeoff

  A major difficulty is trying to approach the optimal accuracy/complexity tradeoff curve, especially early in the top-down approximation process. This is hard because the local estimates only give fuzzy knowledge of the target function. Perhaps an adaptive, recursive estimation strategy could be devised that would improve this knowledge.

- general techniques for providing local estimates

  In the discussions in this dissertation, the applications of the top-down algorithm used *ad hoc* techniques to provide local estimates to target functions. Current investigations are under way to find general, automatic methods for obtaining local estimates for a wide variety of target functions.

References

[1] Barghiel, Cristin, "Feature Oriented Composition of B-Spline Surfaces," Master's Thesis, U. of Waterloo, March, 1994.

[2] Bartels, Richard H., John C. Beatty and Brian A. Barsky, *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann, Los Altos, CA, 1987.

[3] Berman, Deborah F., Jason T. Bartell and David H. Salesin, "Multiresolution Painting and Compositing," *Computer Graphics* (SIGGRAPH '94 Proceedings), July, 1994, pp 85-90.

[4] Casale, Malcolm S., "Free-Form Solid Modeling with Trimmed Surface Patches," *IEEE Computer Graphics & Applications,* Vol. 7, No. 1, January, 1987, pp 33-43.

[5] Catmull, Edwin E. and James H. Clark, "Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes," *Computer Aided Design*, Vol. 10, No. 6, November, 1978, pp 350-355.

[6] Chui, Charles K., *An Introduction to Wavelets*, Academic Press, San Diego, 1992.

[7] Cohen, Elain, Tom Lyche and Richard Riesenfeld, "Discrete B-Splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics," *Computer Graphics and Image Processing*, Vol. 14, No. 2, October, 1980, pp 87-111.

[8] Daubechies, Ingrid, *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[9] de Boor, Carl, "On Calculating with B-splines," *Journal of Approximation Theory*, Vol. 6, No. 1, July, 1972, pp 50-62.

[10] de Casteljau, Paul de Faget, *Shape Mathematics and CAD*, Kogan Page, London, 1986.

[11] DeRose, Tony D., Ronald N. Goldman and J. Michael Lounsbery, "A Tutorial Introduction to Blossoming," in: H. Hagen and D. Roller, eds., *Geometric Modeling, Methods and Applications*, Springer-Verlag, 1991, 267–286.

[12] Duchaineau, Mark A., "World," Independent Study Project, C.L. Morgan Supervising, California State University, Hayward, 1986.

[13] Duchaineau, Mark A., "Construction of Crease-Free Fractals Using Dyadic Splines," *Proceedings of COMPUGRAPHICS92*, December, 1992, pp 222-229.

[14] Eck, Matthias, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery and Werner Stuetzle, "Multiresolution Analysis of Arbitrary Meshes," *Computer Graphics* (SIGGRAPH '95 Proceedings), August, 1995, pp 173-182.

[15] Farin, Gerald, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, Second Edition, Academic Press, Boston, 1990.

[16] Finkelstein, Adam and David H. Salesin, "Multiresolution Curves," *Computer Graphics* (SIGGRAPH '94 Proceedings), July, 1994, pp 261-268.

[17] Forsey, David R., "Motion Control and Surface Modeling of Articulated Figures in Computer Animation," PhD Thesis, U. of Waterloo, September, 1990.

[18] Forsey, David R. and Richard H. Bartels, "Hierarchical B-Spline Refinement," *Computer Graphics* (SIGGRAPH '88 Proceedings), Vol. 22, No. 4, August, 1988, pp 205-212.

[19] Forsey, David R. and Richard H. Bartels, "Surface Fitting with Hierarchical Splines," *ACM Transactions on Graphics*, Vol. 14, No. 2, April, 1995, pp 134-161.

[20] Fournier, Alain, D. Fussell and Loren C. Carpenter, "Computer Rendering of Stochastic Models," *Communications of the A.C.M.*, Vol. 25, 1982, 371-384.

[21] Golub, Gene H. and Charles F. Van Loan, *Matrix Computations* (second ed.), The Johns Hopkins University Press, Baltimore, 1989.

[22] Gortler, Steven J., Peter Schröder, Michael F. Cohen and Pat Hanrahan, "Wavelet Radiosity," *Computer Graphics* (SIGGRAPH '93 Proceedings), August, 1993, pp 221-230.

[23] Hoppe, Hugues, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer and Werner Stuetzle, "Piecewise Smooth Surface Reconstruction," *Computer Graphics* (SIGGRAPH '94 Proceedings), July, 1994, pp 295-302.

[24] Jawerth, B., M.L. Hilton and T.L. Hunstberger, "Local Enhancement of Compressed Images," *J. of Math. Imag. Vision*, Vol. 3, 1993, pp 39-49.

[25] Lounsbery, John Michael, "Multiresolution Analysis for Surfaces of Arbitrary Topological Type," PhD Thesis, U. of Washington, 1994.

[26] Mallat, Stephane G., "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 7, July, 1989, pp 674-693.

[27] Mandelbrot, Benoit B., *The Fractal Geometry of Nature*, W. H. Freeman and Company, New York, NY, 1983.

[28] Mandelbrot, Benoit B., "Fractal Landscapes Without Creases and With Rivers," in *The Science of Fractal Images*, H.-O. Peitgen and D. Saupe Eds., Springer Verlag, New York, 1988, pp 243-260.

[29] Miller, Gavin S. P., "The Definition and Rendering of Terrain Maps," *Computer Graphics* (SIGGRAPH '86 Proceedings), Vol. 20, No. 4, August, 1986, pp 39-48.

[30] Miller, Mark C., "Terrain Representation and Compression," *Pax River Quarterly Report* (3Q92), Advanced Graphics and Parallel Systems, Los Alamos National Laboratory, New Mexico, 1992.

[31] Moore, Ramon E., *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979.

[32] Ramshaw, Lyle, "Blossoming: A Connect-the-Dots Approach to Splines," Research Report 19, Digital Equipment Corp. Systems Research Center, Palo Alto, CA, June, 1987.

[33] Samet, Hanan, *Applications of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[34] Saupe, D., Algorithms for Random Fractals, in *The Science of Fractal Images*, H.-O. Peitgen and D. Saupe Eds., Springer Verlag, New York, 1988, pp 71-112.

[35] Stollnitz, Eric J., Tony D. DeRose and David H. Salesin, "Wavelets for Computer Graphics: A Primer, Part 2," *IEEE Computer Graphics & Applications,* Vol. 15, No. 4, July, 1995, pp 75-85.

[36] Stoneking, Stan, "Screen-Oriented Surface Sculpting Tools," Master's Thesis, Advisor: Kenneth I. Joy, University of California, Davis, June, 1993.

[37] Warnock, John E., "A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures," TR 4–15, NTIS AS-733 671, Computer Science Department, University of Utah, 1969.