A Fresh Look at Generalized Sampling

Diego Nehab
Instituto Nacional de Matemática Pura e Aplicada (IMPA)

Hugues Hoppe
Microsoft Research
# Contents

1 Introduction 2

2 Background 8
   2.1 Reconstruction kernels .................................. 8
   2.2 Analysis filters ........................................... 10

3 Basic notation, definitions, and properties 12

4 Fundamental algorithms 18
   4.1 Interpolation ............................................... 19
   4.2 Inverse discrete convolution ............................... 20
   4.3 Orthogonal projection ...................................... 22
   4.4 Oblique projection ......................................... 25

5 Translation and scaling 27
   5.1 Translation of discretized signals ......................... 27
   5.2 Scaling of discretized signals ............................. 29

6 Approximation of derivatives 37

7 Generalized prefiltering and estimator variance 41

8 Practical considerations 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Grid structure</td>
<td>47</td>
</tr>
<tr>
<td>8.2 Efficient use of piecewise-polynomial kernels</td>
<td>48</td>
</tr>
<tr>
<td>8.3 Prefiltering, reconstruction, and color spaces</td>
<td>50</td>
</tr>
<tr>
<td>8.4 Range constraints</td>
<td>51</td>
</tr>
<tr>
<td>9 Theoretical considerations</td>
<td>53</td>
</tr>
<tr>
<td>10 Experiments and analyses</td>
<td>57</td>
</tr>
<tr>
<td>11 Conclusions</td>
<td>68</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>69</td>
</tr>
<tr>
<td>Appendices</td>
<td>70</td>
</tr>
<tr>
<td>A Source-code</td>
<td>71</td>
</tr>
<tr>
<td>Bibliography</td>
<td>81</td>
</tr>
</tbody>
</table>
Abstract

Discretization and reconstruction are fundamental operations in computer graphics, enabling the conversion between sampled and continuous representations. Major advances in signal-processing research have shown that these operations can often be performed more efficiently by decomposing a filter into two parts: a compactly supported continuous-domain function and a digital filter. This strategy of “generalized sampling” has appeared in a few graphics papers, but is largely unexplored in our community. This survey broadly summarizes the key aspects of the framework, and delves into specific applications in graphics. Using new notation, we concisely present and extend several key techniques. In addition, we demonstrate benefits for prefiltering in image downscaling and supersample-based rendering, and analyze the effect that generalized sampling has on the noise due to Monte Carlo estimation. We conclude with a qualitative and quantitative comparison of traditional and generalized filters.
Many topics in computer graphics involve digital processing of continuous-domain data, so it is unsurprising that discretization and reconstruction are essential operations. Figure 1.1 shows the traditional sampling and reconstruction pipeline. During discretization (e.g., rasterization of a scene, or capture of a digital photograph), a continuous input signal $f$ is passed through an

![Figure 1.1:](image)

Figure 1.1: The traditional signal-processing pipeline is divided into two major stages: discretization and reconstruction. During discretization, a continuous input signal $f$ is convolved with the reflection $\psi^\vee$ of a given analysis filter $\psi$. The resulting prefiltered signal $f_\psi = f * \psi^\vee$ is then uniformly sampled into a discrete sequence $[f_\psi]$. To obtain the output approximation $\tilde{f}$, the reconstruction stage computes the mixed convolution between $[f_\psi]$ and a given reconstruction kernel $\varphi$, i.e., a sum of shifted copies of $\varphi$, where each shifted copy scaled by the corresponding entry in $[f_\psi]$. (Our notation is explained in greater depth in section 3.)
Prefiltering \( f \ast \psi = \beta^0 \)

Sampling \( f \psi \times \mathbb{I} = \mathcal{J} f \psi \mathcal{K} \)

Reconstruction \( \mathcal{J} f \psi \mathcal{K} \ast \varphi = \beta^1 = \tilde{f} \)

Figure 1.2: A continuous function \( f \) is prefiltered with analysis kernel \( \psi \) (here the box function \( \beta^0 \), not to scale). The resulting signal \( f \psi \) is sampled into a discrete sequence \( \mathcal{J} f \psi \mathcal{K} \). The final output \( \tilde{f} \) is obtained by mixed convolution between the discrete sequence \( \mathcal{J} f \psi \mathcal{K} \) and the reconstruction kernel \( \varphi \) (here the hat function \( \beta^1 \), not to scale).

**analysis filter** \( \psi \) (a.k.a. *sampling kernel*, *prefilter*, or *antialiasing filter*) before being sampled. The result is a discrete sequence \( \mathcal{J} f \psi \mathcal{K} \) (e.g., an image). During reconstruction (e.g., interpolation of a texture, or display of an image on a screen), the continuous approximation \( \tilde{f} \) of the original signal is obtained by mixed convolution with a *reconstruction kernel* \( \varphi \) (a.k.a. *generating function*, *basis function*, or *postfilter*). Figure 1.2 illustrates each step of the process with a concrete example in 1D.

The roles of the analysis filter \( \psi \) and reconstruction kernel \( \varphi \) are traditionally guided by the sampling theorem [Shannon, 1949]. Given a sampling rate \( 1/T \), the analysis filter \( \psi = \text{sinc}(\cdot/T) \) eliminates from the input signal \( f \) those frequencies higher than or equal to \( 1/2T \) so that the bandlimited \( f \psi \) can be sampled without aliasing. And in that case, the reconstruction kernel \( \varphi = \text{sinc}(\cdot/T) \) recreates \( \tilde{f} = f \psi \) exactly from the samples.

Sampling may also be interpreted as the problem of finding the function \( \tilde{f} \) that minimizes the norm of the residual \( \| f - \tilde{f} \|_{L^2} \). If we restrict our attention
the space of bandlimited functions, the ideal prefilter is still $\psi = \text{sinc}(\cdot / T)$. However, functions are often not bandlimited in practice (e.g., due to object silhouettes, shadow boundaries, vector outlines, detailed textures), and for efficiency we desire $\psi$ and $\varphi$ to be compactly supported.

In addressing these concerns, the signal-processing community has adopted a generalization of the sampling and reconstruction pipeline [Unser, 2000]. The idea is to represent the prefilter and reconstruction kernels as mixed convolutions of compactly supported kernels and digital filters. As shown in Figure 1.3, digital filters $p$ and $r$ respectively modify the prefilter $\psi$ and the reconstruction kernel $\varphi$. The additional degrees of freedom and effectively larger filter support enabled by $p$ and $r$ allow the design of generalized kernels with better approximation properties or sharper frequency response. Figure 1.4 compares a traditional piecewise cubic kernel (the Catmull-Rom spline, or Keys cubic) with a generalized cubic kernel (the cardinal cubic B-spline).
Figure 1.5: Generalized sampling adds a digital filtering stage to the pipeline. The output \( f_\psi \) of the sampling stage is convolved with a digital transformation filter \( q = p * r \). It is the result \( c \) of this stage (and not \( f_\psi \)) that is convolved with the reconstruction kernel \( \varphi \) to produce the output signal.

Equivalently, the digital filters \( p \) and \( r \) can be combined as \( q = p' * r \) into a separate filter stage as shown in figure 1.5. The result \( f_\psi \) of the sampling stage is transformed by the digital filter \( q \) (a.k.a. correction or basis change) to form a new discrete sequence \( c \), which is then convolved with \( \varphi \) as usual to reconstruct \( \tilde{f} \). The key to the efficiency of this generalized sampling framework is that the digital filters \( p \) and \( r \) that arise in practice are typically compact filters or their inverses [Unser et al., 1991], both of which are parallelizable on multicore CPU and GPU architectures [Ruijters et al., 2008, Nehab et al., 2011]. Thus, the correction stage adds negligible cost.

An important motivation for generalized sampling is improved interpolation [Blu et al., 1999]. As demonstrated in figure 1.6, an image \( f_\psi \) is processed by a digital filter \( q \) resulting in a coefficient array \( c \) which can then be efficiently reconstructed with a simple cubic B-spline filter \( \beta^3 \). The resulting interpolation is sharper and more isotropic (i.e., has higher quality) than that produced by the popular Mitchell-Netravali filter [1988], even though both filters have the same degree and support. The implementation of the digital filtering stage is described in detail in section 4.2. The theory of image upscaling is described in section 5.2 with implementation notes in section 8.2.

Source-code is provided in appendix A.

In graphics, careful prefiltering is often necessary to prevent aliasing. McCool [1995] describes an early application of generalized sampling, in which rendered triangles are antialiased analytically by evaluating a prism spline prefilter. The resulting image is then convolved with a digital filter. In this work, we apply generalized sampling to image downscaling and in general
Introduction

Figure 1.6: Reconstruction example. The top row shows the result of the traditional cubic Mitchell-Netravali filter $M$. The bottom row uses the generalized sampling approach, first applying a digital filter $r = \left[\frac{1}{6}, \frac{2}{6}, \frac{1}{6}\right]$ as a preprocess, and then reconstructing with the cubic B-spline $\beta^3$ — which is less expensive to evaluate on a GPU than filter $M$.

Our aim is to present a concise overview of the major developments in generalized sampling and to extend these techniques to prefiltering in graphics. To facilitate exposition and exploration, we develop a new concise notation for sampling. With this parameter-free notation, key techniques are derived using simple algebraic manipulation. We conclude by comparing a variety of
Figure 1.7: Prefiltering example. The top row shows the result of rendering with the Keys (Catmull-Rom) prefilter $K$. The bottom row shows rendering using a B-spline $\beta^3$, followed by convolution with a digital filter $p^\gamma = \left[ \frac{1}{6}, \frac{4}{6}, \frac{1}{6} \right]$. The generalized prefilter $p * \beta^3$ equals the cubic cardinal B-spline $\beta^3_{\text{int}}$. Kernels $K$ and $\beta^3$ have the same support, but the improved frequency response of $\beta^3_{\text{int}}$ reduces aliasing while maintaining sharpness. (Our notation is explained in section 3.)

Traditional and generalized filters, using frequency-domain visualizations and empirical experiments using both $L_2$ and SSIM metrics, to identify the best strategies available.
2

Background

2.1 Reconstruction kernels

The search for finite-support alternatives to the ideal reconstruction filter has led to many non-polynomial windowed-sinc variants [Meijering et al., 1999a]. The most popular in graphics is the Lanczos window [Duchon, 1979]. (According to our experiments in section 10, the Hamming window [Hamming, 1977] performs better.)

Polynomial functions have been shown to have an efficiency advantage, and to match (and surpass) windowed-sinc approximations in quality [Meijering et al., 2001]. We therefore focus on piecewise polynomial kernels. Like [Thévenaz et al., 2000], we use a set of properties that characterize reconstruction kernels to guide us through the bibliography. For an alternative, chronological survey, please refer to [Meijering, 2002].

The degree $N$ of a kernel $\varphi$ is the maximum degree of its polynomial pieces. The support $W$ of $\varphi$ is the width of the smallest interval outside of which $\varphi$ vanishes (assuming a sample spacing of one). Increasing either the degree or support of a kernel $\varphi$ introduces additional degrees of freedom for the design of good kernels, but unfortunately also adds to the runtime computational cost.
2.1. Reconstruction kernels

Most kernels are symmetric about the origin. The regularity \( R \) measures the smoothness of \( \phi \). That is, a kernel \( \phi \) is said to be in \( C^R \) if it is differentiable \( R \) times. The space of functions spanned by a generator \( \phi \) is denoted by \( V_\phi \). The order of approximation \( L \) of \( V_\phi \) measures the rate at which the residual between \( f \) and its optimal approximation \( \tilde{f}_T \in V_\phi \) vanishes as the sample spacing \( T \) is reduced:

\[
\| f - \tilde{f}_T \|_{L^2} = C f T^L \quad \text{as} \quad T \to 0.
\]

Equivalently, a space with approximation order \( L \) can reproduce polynomial signals of up to degree \( L - 1 \) \cite{Strang_and_Fix_1973}. Enforcement of regularity, symmetry, and approximation order in \( \phi \) consume degrees of freedom from the design process. In fact, the best approximation order a kernel of degree \( N \) can attain is \( L = N + 1 \). This optimal order is achievable even with a compact support \( W = N + 1 \) \cite{Blu_et_al_2001}. Various strategies for setting the remaining degrees of freedom have led to the development of a multitude of reconstruction kernels.

Mitchell and Netravali \cite{1988} design a family of cubic kernels by starting with \( W = 4, R = 1, \) and \( L = 1 \). They set the two remaining degrees of freedom by subjectively evaluating the amount of ringing, blur, and anisotropy in upsampled images. Alternatively, setting \( L = 2 \) leaves only one degree of freedom, and they similarly set its value based on subjective quality.

**Interpolating kernels** A reconstruction kernel \( \phi \) is interpolating if it satisfies \( \phi(0) = 1 \) and \( \phi(k) = 0, k \in \mathbb{Z} \setminus \{0\} \). Naturally, enforcing this property further eliminates degrees of freedom. Popular interpolating kernels include the ubiquitous nearest-neighbor (or box, \( L = 1, W = 1 \)) and linear (or hat, \( L = 2, W = 2 \)) interpolation kernels. These are members of the family of local Lagrangian interpolators, which have optimal order and minimum support \( L = W = N + 1 \) \cite{Schafer_and_Rabiner_1973, Schaum_1993, Blu_et_al_2001} but offer no regularity. Several authors have reached the Catmull-Rom cubic spline \cite{Catmull_and_Rom_1974} by different means \cite{Keys_1981, Park_and_Schowengerdt_1983, Meijering_et_al_1999b, Blu_et_al_2001}. It is the only \( C^1 \)-continuous cubic interpolating kernel with optimal order and minimum support.

Other noteworthy kernels include an additional cubic (\( W = 6, L = 4 \)) by Keys \cite{1981}, the \( C^0 \)-continuous (interpolating) and the \( C^1 \)-continuous
Background

(not interpolating) quadratics \((N = 2, W = 3, L = 2)\) of [Dodgson 1997], a quartic \((L = 5, W = 7, R = 1)\) by [German 1997], and the quintic and septic kernels \((W = 6 \text{ and } 8, \text{ but } L = 3)\) by [Meijering et al. 1999b]. The support of these kernels is larger than necessary for their approximation order.

Generalized kernels  A breakthrough came from the idea that the responsibility for interpolating the original samples need not be imposed on the continuous kernel \(\varphi\) itself, but can instead be achieved by using a digital correction filter \(q\). This was first demonstrated in the context of B-spline interpolation [Hou and Andrews 1978, Unser et al. 1991].

B-splines are the most regular members of a class of functions called MOMS (for Maximal Order, Minimum Support) [Blu et al. 2001]. The best performing kernels, the O-MOMS (Optimal MOMS), trade off regularity to minimize the leading coefficient \(C_L\) in the optimal mean approximation error
\[
\| f - \tilde{f}_T \|_{L_2} = C_L T^L \| f^{(L)} \|_{L_2} \quad \text{as} \quad T \to 0.
\]
If continuous derivatives are desired, the SO-MOMS (Sub-Optimal MOMS) minimize coefficient \(C_L\) subject to \(C_1\)-continuity.

It is possible to further reduce reconstruction error by mimicking the low-frequency behavior of orthogonal projection (see below). The shifted linear interpolation scheme of [Blu et al. 2004] gives up on symmetry and uses the additional freedom to minimize the approximation error. Quasi-interpolation schemes give up on interpolation of \([f_\varphi]\), so that \(q\) is freed of this restriction. In that case, the interpolation property holds only when \(f\) is a polynomial of degree less than \(L\) (the quasi-interpolation order). [Blu and Unser 1999a] describe an IIR design for \(q\), [Condat et al. 2005] an all-pole design, and [Dalai et al. 2006] a FIR design. The improvements due to this relaxation are often substantial, particularly for low-degree schemes \((N \leq 3)\).

2.2 Analysis filters

As explained earlier, an important goal is orthogonal projection: minimizing the residual \(\| f - \tilde{f} \|_{L_2}\) between the reconstruction \(\tilde{f}\) and the input signal \(f\). Given a reconstruction kernel \(\varphi\), orthogonal projection is achieved using a prefilter known as the dual of \(\varphi\) and denoted by the symbol \(\hat{\varphi}\). This prefilter may be written in the form \(p * \varphi\) (see section 4.3). Exploiting the fact that in
computer graphics we often have access to the input signal prior to sampling, Kajiya and Ullner [1981] use this orthogonal projection for antialiased text rasterization. In their work, $\varphi$ is a Gaussian that models the CRT electron beam, and $f$ corresponds to the text being rendered. They further restrict the optimization to non-negative coefficients $c$ and explore a perceptual alternative to the $L_2$ norm.

This idea was extended by McCool [1995] to include other prefilters of the form $p \ast \psi$, where $\psi$ is a compactly supported B-spline basis function. A similar factorization is used: first the input is prefiltered with $\psi$, then the result is transformed to what would be obtained if directly prefiltering with the cardinal spline.

Orthogonal projection is commonly used in processing of previously discretized input signals. Most image processing operations result in signals outside of the approximation space. These can then be projected back. In the case of scaling and translation, there are efficient algorithms to achieve this [Unser et al. 1995a,b, Muñoz et al., 2001].

The approach of orthogonal projection has not been widely adopted in computer graphics. In part, there is a lack of familiarity with the framework. But also, there is the issue that the perceptual quality of an approximation involves a subjective balance between aliasing, blurring, ringing, and positivity, to which the $L_2$ norm is oblivious (as noted by Kajiya and Ullner [1981]). Some favor the sharpness offered by filters with negative lobes (the “negative lobists” [Blinn 1989]), while others fear the accompanying ringing in dark regions (a problem accentuated by gamma correction). Rendering systems therefore offer several alternatives, including box, hat, Mitchell-Netravali, Catmull-Rom (Keys), Gaussian, and windowed-sinc filters [Pixar 2005], but rarely (if ever) offer orthogonal or oblique projections.
Many of the derivations in generalized sampling involve tedious manipulations of integrals and summations, including changes of index variables. In this section, we introduce a new index-free notation to help eliminate this problem. Using this notation and a small set of properties, we are able to simplify the derivations of many algorithms and concepts to trivial algebraic manipulations.

For simplicity, we formulate the mathematics in one dimension. However, all results are easily extended to 2D images and 3D volumes in a separable way by making use of tensor-product basis functions. Furthermore, we assume all signals are real even though results are easily generalized to complex signals.

We denote the implicit argument to a univariate function with a dot “·”:

\[ f(\cdot + k) \overset{\text{def}}{=} x \mapsto f(x + k). \tag{3.1} \]

The prefiltering stage in figures 1.1 and 1.5 simply perform a continuous convolution between the signal \( f \) and the prefilter kernel \( \psi \):

\[ f \star \psi \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(t) \psi(\cdot - t) \, dt. \tag{3.2} \]

We denote discrete sequences by bold letters, or write the values explicitly, enclosed in bold square-brackets \([\ ]\):

\[ c \overset{\text{def}}{=} \ldots, c_{-2}, c_{-1}, c_0, c_1, c_2, \ldots \], \quad c_k \in \mathbb{R}, \; k \in \mathbb{Z}. \tag{3.3} \]
An element of a sequence can be selected by a subscript so that \( c_k \overset{\text{def}}{=} c_k \) above.

A key part of our new notation is the use of bold square-brackets, when enclosing an expression with a single free variable, to denote the discrete sequence that results from uniformly sampling the expression at all multiples of a sampling spacing \( T \):

\[
[f]_T \overset{\text{def}}{=} \left[ \ldots, f(-2T), f(-T), f(0), f(T), f(2T), \ldots \right]
\]

\[
\overset{\text{def}}{=} [f], \text{ in short when } T = 1.
\]

This is the operation performed by the sampling stage of figures 1.1–1.5.

The digital filtering introduced by the generalized sampling pipeline in figure 1.5 corresponds to discrete convolution with the discrete sequence \( q \):

\[
c = b \ast q, \text{ where } c_k = \sum_{i \in \mathbb{Z}} b_i q_{k-i}, k \in \mathbb{Z}.
\]

The reconstruction stage in figures 1.1–1.5 shares many properties with convolutions. We therefore introduce notation for a mixed convolution with spacing \( T \) between a discrete sequence \( c \) and a reconstruction kernel \( \varphi \):

\[
c \ast_T \varphi \overset{\text{def}}{=} \sum_{k \in \mathbb{Z}} c_k \varphi(\cdot - kT) \quad \text{and in particular } \ c \ast \varphi \overset{\text{def}}{=} c \ast_1 \varphi.
\]

In most cases, commutativity and associativity apply to all combinations of convolutions. Parentheses are unnecessary as there is no danger of ambiguity. We can manipulate convolutions like products:

\[
(b \ast c) \ast f = b \ast (c \ast f) \quad \text{and} \quad (f \ast g) \ast b = f \ast (g \ast b).
\]

The only exception is in expressions involving multiple mixed convolutions with different spacings. Even then, it is still true that

\[
b \ast_T (c \ast_S f) = c \ast_S (b \ast_T f),
\]

but the spacings must be the same for us to factor out the discrete convolution:

\[
b \ast_T (c \ast_T f) = c \ast_T (b \ast_T f) = (b \ast c) \ast_T f.
\]

For repeated convolution, we use a notation reminiscent of exponentiation:

\[
f^{\ast n} = f \ast \cdots \ast f \quad \text{and} \quad b^{\ast n} = b \ast \cdots \ast b
\]
The reflection of sequences and functions is denoted by
\[ (c^\vee)_k \overset{\text{def}}{=} c_{-k}, \quad k \in \mathbb{Z} \quad \text{and} \quad f^\vee(x) \overset{\text{def}}{=} f(-x), \quad x \in \mathbb{R} \] (3.12)
and distributes over all flavors of convolution:
\[ (b * c)^\vee = b^\vee * c^\vee, \quad (f * g)^\vee = f^\vee * g^\vee, \quad \text{and} \quad (b * \varphi)^\vee = b^\vee * \varphi^\vee. \] (3.13)

The simple connection between inner-products and convolutions,
\[ \langle f, \psi \rangle \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(t) \psi(t) \, dt = (f * \psi^\vee)(0), \] (3.14)
is the source for the pervasive reflection operation. Its generalization
\[ \langle f, \psi(\cdot - k) \rangle \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(t) \psi(t-k) \, dt = (f * \psi^\vee)(k) \] (3.15)
lets us express in compact form the fundamental operation of sampling a function \( f \) with an analysis filter \( \psi \):
\[ \ldots, \langle f, \psi(\cdot - 1) \rangle, \langle f, \psi \rangle, \langle f, \psi(\cdot + 1) \rangle, \ldots = [f * \psi^\vee]. \] (3.16)

Using our notation, we can succinctly express the key property that sampling a mixed convolution with matching spacing \( T \) is equivalent to performing a discrete convolution between the sequence and the sampled function:
\[ [b *_T f]_T = b * [f]_T. \] (3.17)
In other words, these operations commute.

The continuous unit impulse \( \delta \) (the Dirac delta) and the discrete unit impulse \( \delta \) (the Kronecker delta) are the identity elements for the convolution operations:
\[ \delta * f \overset{\text{def}}{=} f, \quad \forall f \quad \text{and} \quad \delta * c \overset{\text{def}}{=} c, \quad \forall c. \] (3.18)

The discrete impulse can also be defined simply as
\[ \delta = [\ldots, 0, 0, 1, 0, 0, \ldots] \quad \text{where} \quad \delta_0 = 1. \] (3.19)

The discrete convolution operation \( c = b * q \) can often be efficiently inverted (see section 4.2). The inverse operation is again a discrete convolution:
\[ b = c * q^{-1} \quad \text{with} \quad q * q^{-1} = \delta. \] (3.20)
Reflection and convolution-inverse commute, so we can define:

\[ b^\forall \overset{\text{def}}{=} (b^{-1})^\forall = (b^\forall)^{-1}. \] (3.21)

Interestingly, the derivative operation does not distribute over convolution:

\[ (f * g)' = f' * g = f * g'. \] (3.22)

This rule lets us express the derivative operation as a convolution, since

\[ f = f * \delta \implies f' = (f * \delta)' = f * \delta'. \] (3.23)

For higher-order derivatives, we use the short notation

\[ f^{(n)} = f * (\delta')^n. \] (3.24)

The discrete analog of the derivative is the finite difference. We can express the (backward) differencing operation as convolution with the sequence

\[ \Delta = [\ldots, 0, 1, -1, 0, \ldots] \quad \text{where} \quad \Delta_0 = 1. \] (3.25)

Just as with the derivative operation,

\[ \Delta * (b * c) = (\Delta * b) * c = b * (\Delta * c). \] (3.26)

The continuous unit (or Heaviside) step function \( u \) is the integral of the unit impulse. Accordingly, the discrete unit step function \( u \) is the summation of the discrete unit impulse:

\[ u(x) = \int_{-\infty}^{x} \delta(t) \, dt \quad \text{and} \quad u_i = \sum_{k=-\infty}^{i} \delta_k. \] (3.27)

From these definitions, it is clear that

\[ (f * u)(x) = \int_{-\infty}^{x} f(t) \, dt \quad \text{and} \quad (c * u)_i = \sum_{k=-\infty}^{i} c_k. \] (3.28)

Furthermore,

\[ u * \delta' = \delta \quad \text{and} \quad u * \Delta = \delta, \] (3.29)

so that the pairs \( u, \delta' \) and \( u, \Delta \) are convolution inverses.
We use $\tau_h$ to denote the translation of the impulse $\delta$ by an offset $h$:

$$\tau_h = \delta(\cdot - h). \quad (3.30)$$

This lets us express the translation of a function $f$ as a convolution:

$$f(\cdot - h) = \tau_h \ast f. \quad (3.31)$$

The centered B-spline basis functions $\beta^n$ are an important family of generating functions. The box filter $\beta^0$ can be defined as:

$$\beta^0 = \Delta \ast u \ast \tau_{-1/2}. \quad (3.32)$$

The hat filter $\beta^1$ and the remaining B-splines are recursively defined as:

$$\beta^n = \beta^{n-1} \ast \beta^0, \quad (3.33)$$

or equivalently using the notation for repeated convolution:

$$\beta^n = (\beta^0)^{(n+1)} = \Delta^{(n+1)} \ast u^{(n+1)} \ast \tau_{(n+1)/2}. \quad (3.34)$$

Note that the one-sided power function can be written equivalently as

$$u^{(n+1)}(x) = \frac{x^n}{n!} u(x). \quad (3.35)$$

A useful shorthand for the cross-correlation between two functions is:

$$a_{\varphi,\psi} \overset{\text{def}}{=} \int_{-\infty}^{\infty} \varphi(t) \psi(t - \cdot) \, dt = \varphi \ast \psi'. \quad (3.36)$$

In particular, we denote the auto-correlation as $a_\varphi \overset{\text{def}}{=} a_{\varphi,\varphi}$.

Two functions $\varphi$ and $\psi$ are biorthogonal if they satisfy

$$\langle \varphi(\cdot - i), \psi(\cdot - j) \rangle = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise}, \end{cases} \quad (3.37)$$

where $i, j \in \mathbb{Z}$, or equivalently in our notation,

$$[a_{f,g}] = \delta. \quad (3.38)$$

The Discrete Time Fourier Transform (DTFT) of a sequence and the Fourier Transform of a function are defined respectively by

$$\hat{c}(\xi) \overset{\text{def}}{=} \sum_{k \in \mathbb{Z}} c_k e^{-2\pi i k} \quad \text{and} \quad \hat{f}(\xi) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(x) e^{-2\pi i x} \, dx. \quad (3.39)$$
Convolution in the time domain becomes a product in the frequency domain. This is also true of mixed convolutions:

\[
\hat{(b * c)} = \hat{c} \hat{b}, \quad \hat{(f * g)} = \hat{f} \hat{g}, \quad \text{and} \quad \hat{(f * c)} = \hat{f} \hat{c}.
\]  

(3.40)

In our notation, the impulse train (Dirac comb) is a mixed convolution:

\[
\Pi \overset{\text{def}}{=} \sum_{k \in \mathbb{Z}} \delta(\cdot - k) = [1] * \delta = 1 * \delta
\]  

(3.41)

It has the important property of being its own Fourier transform:

\[
\hat{\Pi} = \Pi.
\]  

(3.42)

See table 3.1 for a list of properties and constructions expressed using the notation we just presented.

**Table 3.1:** Some properties and constructions using the notation presented in this section.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Expressed in our notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry of $\varphi$</td>
<td>$\varphi = \varphi'$</td>
</tr>
<tr>
<td>Interpolation property</td>
<td>$[\varphi] = \delta$</td>
</tr>
<tr>
<td>Unit integral</td>
<td>$\varphi * [1] = 1$</td>
</tr>
<tr>
<td>Partition of unity</td>
<td>$\varphi * [1] = 1$</td>
</tr>
<tr>
<td>Cross-correlation of $\varphi$ and $\psi$</td>
<td>$a_{\varphi,\psi} = \varphi * \psi'$</td>
</tr>
<tr>
<td>Biorthogonality of $\varphi$ and $\psi$</td>
<td>$[a_{\varphi,\psi}] = \delta$</td>
</tr>
<tr>
<td>Auto-correlation of $\varphi$</td>
<td>$a_{\varphi} = \varphi * \varphi'$</td>
</tr>
<tr>
<td>Orthogonality of $\varphi$</td>
<td>$[a_{\varphi}] = \delta$</td>
</tr>
<tr>
<td>Cardinal kernel $\varphi_{\text{int}}$</td>
<td>$[\varphi]^{-1} * \varphi$</td>
</tr>
<tr>
<td>Dual kernel $\hat{\varphi}$</td>
<td>$[a_{\varphi}]^{-1} * \varphi$</td>
</tr>
<tr>
<td>Orthogonal kernel $\phi$</td>
<td>$[a_{\varphi}]^{\frac{1}{2}} * \varphi$</td>
</tr>
</tbody>
</table>
Using the new notation introduced in section 3, we now describe some fundamental algorithms in generalized sampling. While none of these algorithms are novel, the new notation allows more concise (and arguably more intuitive) derivation.

The algorithms in the next sections address different scenarios based on the selection of prefilter and reconstruction kernels:

- **In interpolation**, the input is a sampled representation assumed to be obtained without prefiltering, or equivalently by using a prefilter $\psi$ equal to the unit impulse $\delta$. Therefore the goal is to find the unique reconstruction $\tilde{f}$ that interpolates the samples — among functions in the space $V_\varphi$ of the reconstruction kernel $\varphi$.

- The technique of **orthogonal projection** addresses the case where the prefilter $\psi$ spans the same function space as the reconstruction kernel $\varphi$, e.g. the space of piecewise cubic B-splines. The approach is to minimize the $L_2$ difference between the input signal and the reconstruction.

- **Finally, oblique projection** considers the case where the prefilter $\psi$ and reconstruction kernel $\varphi$ span different function spaces. It sets the reconstruction error to be orthogonal to the prefiltering space.
Table 4.1: Fundamental algorithms and their associated digital filters.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Prefilter</th>
<th>Reconstruction</th>
<th>Digital filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation</td>
<td>$\psi = \delta$</td>
<td>$\varphi$</td>
<td>$q = [\varphi]^{-1}$</td>
</tr>
<tr>
<td>Orthogonal projection</td>
<td>$\psi = \varphi$</td>
<td>$\varphi$</td>
<td>$q = [a\varphi]^{-1}$</td>
</tr>
<tr>
<td>Oblique projection</td>
<td>$\psi$</td>
<td>$\varphi$</td>
<td>$q = [\varphi * \psi^p]^{-1}$</td>
</tr>
</tbody>
</table>

Table 4.1 summarizes the digital filter associated with each of the three approaches, as derived in the next sections. Interestingly, both interpolation and orthogonal projection can be seen as special cases of oblique projection.

4.1 Interpolation

When the prefiltering process is unknown (or absent if $\psi = \delta$), it is often desired that the reconstructed function $\tilde{f}$ interpolate the sample values $[f_{\psi}]$. With traditional sampling (figure 1.1), this interpolation property requires:

$$[f_{\psi}] = [\tilde{f}]$$  \hspace{1cm} (4.1)
$$= [f_{\psi} * \varphi]$$  \hspace{1cm} (4.2)
$$= [f_{\psi}] * [\varphi].$$  \hspace{1cm} (4.3)

As $[f_{\psi}]$ is arbitrary, an interpolating reconstruction kernel $\varphi$ must therefore satisfy

$$[\varphi] = \delta.$$  \hspace{1cm} (4.4)

As noted earlier, these interpolation constraints may severely hinder the design of a reconstruction kernel with good approximation properties. Therefore, instead of requiring the kernel itself to satisfy the interpolation property, generalized sampling (figure 1.5) introduces a digital filtering stage for that purpose. We find an appropriate digital filter for interpolation with an arbitrary kernel $\varphi$ as follows:

$$[f_{\psi}] = [\tilde{f}]$$  \hspace{1cm} (4.5)
$$= [c * \varphi]$$  \hspace{1cm} (4.6)
$$= c * [\varphi] \Rightarrow$$  \hspace{1cm} (4.7)
$$c = [f_{\psi}] * [\varphi]^{-1}. \hspace{1cm} (4.8)$$
In other words,
\[ c = [f_\psi] * q \quad \text{with} \quad q = [\varphi]^{-1}. \] (4.9)

The convolution of the digital filter \( q = [\varphi]^{-1} \) with the kernel \( \varphi \) is called the \textit{cardinal} \( \varphi_{\text{int}} \) (see figure 4.1b):
\[ \tilde{f} = c * \varphi = [f_\psi] * [\varphi]^{-1} * \varphi \] (4.10)
so that
\[ \tilde{f} = [f_\psi] * \varphi_{\text{int}} \quad \text{with} \quad \varphi_{\text{int}} = [\varphi]^{-1} * \varphi. \] (4.11)

We can use (4.4) to verify that \( \varphi_{\text{int}} \) is indeed an interpolating kernel:
\[ \Phi_{\varphi_{\text{int}}} = \begin{bmatrix} \varphi_{\text{int}} \end{bmatrix} = \begin{bmatrix} [\varphi]^{-1} * \varphi \end{bmatrix} = \begin{bmatrix} \varphi \end{bmatrix}^{-1} * \begin{bmatrix} \varphi \end{bmatrix} = \delta. \] (4.12)

The key point is that the generalized sampling approach has the advantage of using the compactly supported \( \varphi \) during reconstruction, when performance is paramount. Since digital filtering with \( q \) is performed during preprocessing, it does not add any computational cost to the reconstruction stage. In the words of Blu et al. [1999], one obtains “higher-quality at no additional cost”. In contrast, the cardinal kernel \( \varphi_{\text{int}} \) has infinite support. Finite approximations have a significantly wider support than \( \varphi \), increasing reconstruction costs.

### 4.2 Inverse discrete convolution

Discrete convolution by a sequence \([\varphi]\) as in (4.7) is a linear operator and therefore expressible in matrix form:
\[ [f_\psi] = c * [\varphi] = \Phi c, \quad \text{for some matrix} \quad \Phi. \] (4.13)

Inverting the process as in (4.9) is more challenging, as it amounts to solving a linear system:
\[ c = [f_\psi] * [\varphi]^{-1} = \Phi^{-1} [f_\psi]. \] (4.14)

Fortunately, typical matrices \( \Phi \) have regular structure: for even-periodic extensions, they are \textit{almost} Toeplitz. We describe an algorithm adapted from Malcolm and Palmer [1974] for solving the common case where \([\varphi] = [p, q, p] \) is symmetric with support width of 3 elements. A similar algorithm appears in Hummel [1983] in the context of orthogonal projection. The tridiagonal matrix \( \Phi \) and its \( LU \)-decomposition are:
4.2. Inverse discrete convolution

The factorization is performed and stored only once, then reused to solve multiple linear systems (one for each column and row in the image). Moreover, when $\Phi$ is diagonally dominant, the factorization is highly compressible, as the sequence $[\ell_0, \ell_1, \ldots]$ quickly converges to a limit value $\ell_\infty$. The last diagonal element $v = 1 + 1/\ell_\infty$ of matrix $U$ needs to be handled separately.

Computing $c = [f_0] * [\varphi]^{-1}$ when $[\varphi]$ has support 3 requires only $3n$ products and $2n$ additions. Coincidentally, this is the same cost as computing $[f_0] = c * [\varphi]$. If one can make do with a scaled version $d = p c$ of the solution, for example by pre-scaling the kernel so that $p = 1$, the computation requires $n$ fewer products.

The forward- and back-substitutions are very simple, and we include full source-code in appendix A (function linear_solve, lines 12–32). For convenience, we also provide the LU-decomposition arising from the cubic B-spline and O-MOMS interpolation problems. The cubic B-spline has $[\varphi] = \frac{1}{6}[1, 4, 1]$ and needs only 8 coefficients before the sequence converges to single-precision floating-point. (See lines 162–164.) For the cubic O-MOMS, $[\varphi] = \frac{1}{21}[4, 13, 4]$ and only 9 coefficients are needed (See lines 192–195.)

Our vectorized, multicore implementation of the same algorithm in 2D runs at 800MiPix/s on an Intel Core i7 980X CPU.

An equivalent approach favored by the signal processing community is to formulate this inverse convolution as a sequence of recursive filters [Hou and Andrews, 1978, Unser et al., 1991]. The difficulty is dealing with the poles in $q = [\varphi]^{-1}$ and the trick is to factor the filter into cascaded causal and anticausal parts, each realized as a stable recursive filter. Interestingly, the operations are almost exactly the same as those described in appendix A. The only difference is in the treatment of elements close to the boundaries. Whereas the recursive-filtering approach conceptually extends the input so as to compute the initial feedback, the LU-factorization simply uses different coefficients for the elements close to the boundary.
Even if $[\varphi]$ has larger support (i.e., $W > 4$), the inverse convolution can still be performed efficiently using $LU$-factorization in a similar way. The matrices $L$ and $U$ have bandwidth $\left\lfloor \frac{W+1}{2} \right\rfloor$, so the forward/back-substitution steps use several feedback elements, much like higher-order causal/anti-causal recursive filters.

In signal processing, higher-order recursive filters are sometimes further factored into chains of first- and second-order recursive filters. This is equivalent to factoring the $L$ and $U$ matrices into products of lower and upper bidiagonal or tridiagonal matrices. However, computations using such factorizations traverse the data additional times and do not save arithmetic operations. Thus, the additional factorization is not beneficial in modern computer architectures where memory access is expensive.

A modern parallelization of recursive filters for GPUs recently achieved over 6GiPix/s on an NVIDIA GTX 480 GPU for the bicubic B-spline interpolation problem [Nehab et al., 2011]. The key message is that, whatever the approach followed or architecture used, the digital processing stage in generalized sampling is extremely fast.

### 4.3 Orthogonal projection

We now derive an algorithm to obtain the orthogonal projection $P_\varphi f$ of the input signal $f$ into the reconstruction space $V_\varphi$. The algorithm computes the coefficient array $c$ expressing this projection as $P_\varphi f = c \ast \varphi$. To compute $c$, we need to determine the appropriate prefilter $\psi$ and digital filter $q$. The orthogonality condition is:

\[
(f - \tilde{f}) \perp V_\varphi \iff \langle f - \tilde{f}, \varphi(\cdot - k) \rangle = 0 \quad \text{for} \quad k \in \mathbb{Z}, \quad (4.15)
\]

\[
\iff \left[(f - \tilde{f}) \ast \varphi'\right] = 0. \quad (4.16)
\]

From there,

\[
\left[(f - \tilde{f}) \ast \varphi'\right] = 0 \Rightarrow \quad (4.17)
\]

\[
\left[f \ast \varphi'\right] = \left[\tilde{f} \ast \varphi'\right] \quad (4.18)
\]

\[
= \left[c \ast \varphi \ast \varphi'\right] \quad (4.19)
\]

\[
= c \ast \left[\varphi \ast \varphi'\right] \Rightarrow \quad (4.20)
\]

\[
c = \left[f \ast \varphi'\right] \ast \left[\varphi \ast \varphi'\right]^{-1}. \quad (4.21)
\]
4.3. Orthogonal projection

In other words, 
\[ c = [f \ast \psi^*] \ast q \quad \text{with} \quad \psi = \varphi \quad \text{and} \quad q = [a_{\varphi}]^{-1}. \quad (4.22) \]

Here, \( a_{\varphi} = \varphi \ast \varphi^* \) is the auto-correlation of \( \varphi \), so that the matrix associated with the linear system is none other than the Gramian matrix reached by Kajiya and Ullner [1981] and Hummel [1983]. It is important to point out that, unless the output device uses \( \varphi \) for reconstruction, it makes little sense to directly use the coefficient array \( c \) for display. It may be necessary to prefilter \( P_{\varphi} f \) for display, or at least sample its reconstruction (see section 10 and figure 10.4):

\[ [P_{\varphi} f] = [c \ast \varphi] = c \ast [\varphi]. \quad (4.23) \]

Recall that the orthogonal projection \( P_{\varphi} \) is equivalent to convolution with the dual \( \check{\varphi} \) (figure 4.1c). To find the expression for \( \check{\varphi} \) and verify that it belongs to \( V_{\varphi} \), note that\(^1\)

\[
\begin{align*}
    c &= [f \ast \varphi^*] \ast [a_{\varphi}]^{-1} = [f \ast \varphi^*] \ast [a_{\varphi}]^{-1} \\
    &= [f \ast \varphi^* \ast [a_{\varphi}]^{-1}] = [f \ast (\varphi \ast [a_{\varphi}]^{-1})].
\end{align*}
\]

\(^1\)Remember that \( a_{\varphi} = \varphi \ast \varphi^* \) is always symmetric.
so that
\[ c = [f * \phi^\vee] \quad \text{with} \quad \phi^\vee = \phi * [a_\phi]^{-1}. \tag{4.24} \]

Thus, the traditional sampling approach would be to set the prefilter \( \psi = \phi^\vee \).
The generalized sampling approach lets us avoid working directly with the dual \( \phi^\vee \), which typically has infinite support. We instead prefilter \( f \) with the more convenient, compactly supported kernel \( \phi \), and convolve with the inverse of the sampled auto-correlation.

As a side note, it is easy to verify the biorthogonality of \( \phi \) and \( \phi^\vee \), i.e.
\[ \langle \phi^\vee(\cdot - i), \phi(\cdot - j) \rangle = \delta_{ij}, \tag{4.25} \]
using our notation:
\[
\begin{align*}
[\phi * \phi^\vee] &= \left[\phi * (\phi * [a_\phi]^{-1})^\vee\right] \\
&= \left[\phi * \phi^\vee * [a_\phi]^{-1}\right] \\
&= \left[\phi * \phi^\vee\right] * [a_\phi]^{-1} \\
&= [a_\phi] * [a_\phi]^{-1} \\
&= \delta.
\end{align*}
\tag{4.26} \tag{4.27} \tag{4.28} \tag{4.29} \tag{4.30}

McCool [1995] explored orthogonal projection in the context of rendering, but may have mistakenly used the correction filter \( q = [\phi]^{-1} * [\phi]^{-1} \) instead of \( q = [a_\phi]^{-1} \) for generalized prefiltering.

**Orthogonal kernels**  
It is easy to show (via the frequency domain) that the ideal low-pass filter sinc equals its own auto-correlation. Because sinc satisfies the interpolation condition, it happens that sinc is also its own dual, i.e., it is an orthogonal kernel. This means that the ideal sampling procedure amounts to the orthogonal projection of function \( f \) into the space of bandlimited functions. Another example of orthogonal kernel is the box filter (its auto-correlation is the hat filter, which also interpolates). In general, other typical kernels are not orthogonal, so that \( \phi \) and \( \phi^\vee \) are quite different.

**Notable equivalent generating functions**  
We have already seen three generating functions for the same approximation space \( V_\phi \): The basic \( \phi \) itself, its cardinal \( \phi_{\text{int}} \), and its dual \( \phi^\vee \). To complete the picture, we now find an
4.4. Oblique projection

orthogonal generating function $\phi$ for $V_\varphi$. This is mostly of theoretical interest. To do so, we numerically compute the “convolution square root” of $[a_\varphi]^{-1}$ via the Fourier series expansion of DTFT $^{-1}/2([a_\varphi])$. Then,

$$\phi = \varphi * [a_\varphi]^{-\frac{1}{2}}$$

where $[a_\varphi]^{-\frac{1}{2}} * [a_\varphi]^{-\frac{1}{2}} = [a_\varphi]^{-1}$, (4.31)

and we can verify that $\phi$ is indeed orthogonal:

$$[\phi * \phi^\vee] = [\varphi * [a_\varphi]^{-\frac{1}{2}} * (\varphi * [a_\varphi]^{-\frac{1}{2}})^\vee]$$

(4.32)

$$= [\varphi * \varphi^\vee] * [a_\varphi]^{-\frac{1}{2}} * [a_\varphi]^{-\frac{1}{2}}$$

(4.33)

$$= [a_\varphi] * [a_\varphi]^{-1}$$

(4.34)

$$= \delta.$$  

(4.35)

Figure 4.1 shows the four bases associated with the cubic B-splines.

4.4 Oblique projection

Unlike in rendering applications, many signal-processing applications typically have little control over the analysis filter $\psi^\vee$ (e.g., it is part of an acquisition device). Moreover, there may be little control over the reconstruction filter $\varphi$ (e.g., it is part of a display device). Naturally, this prevents the use of the orthogonal projection. Instead, given both $\psi^\vee$ and $\varphi$, consistent sampling [Unser and Aldroubi, 1994] is a strategy for obtaining the oblique projection $P_{\varphi,\perp \psi} f$ of a signal $f$ into space $V_\varphi$, where the residual is orthogonal to $V_\psi$ (rather than being orthogonal to $V_\varphi$):

$$[(f - \tilde{f}) * \psi^\vee] = 0 \Rightarrow$$

$$[f * \psi^\vee] = [\tilde{f} * \psi^\vee]$$

(4.36)

$$= [c * \varphi * \psi^\vee]$$

(4.37)

$$= c * [\varphi * \psi^\vee] \Rightarrow$$

$$c = [f * \psi^\vee] * [\varphi * \psi^\vee]^{-1}.$$  

(4.38)

In other words,

$$c = [f_{\psi}] * q \quad \text{with} \quad q = [\varphi * \psi^\vee]^{-1} = [a_{\varphi,\psi}]^{-1},$$

(4.39)

where $a_{\varphi,\psi}$ is the cross-correlation of $\varphi$ and $\psi$. 
An equivalent characterization follows from the projection property. If a signal \( f = c \ast \varphi \) already belongs to \( V_\varphi \), then its oblique projection \( P_{\varphi \perp \psi} f \) must be \( f \) itself (hence, “consistent”):

\[
\begin{align*}
    c \ast \varphi &= P_{\varphi \perp \psi} (c \ast \varphi) \Rightarrow \\
    c &= [c \ast \varphi \ast \psi^*] \ast q = c \ast [\varphi \ast \psi^*] \ast q \Rightarrow \\
    q &= [\varphi \ast \psi^*]^{-1}
\end{align*}
\]

Yet another characterization is that oblique projection selects \( q \) to make the effective analysis filter \( q' \ast \psi \) biorthogonal to the reconstruction kernel \( \varphi \):

\[
\begin{align*}
    [\varphi \ast (q' \ast \psi)^*] &= \delta \Rightarrow \\
    [\varphi \ast q \ast \psi^*] &= \delta \Rightarrow \\
    [\varphi \ast \psi^*] \ast q &= \delta \Rightarrow \\
    q &= [\varphi \ast \psi^*]^{-1}
\end{align*}
\]

The approximation error of the oblique projection is bounded by

\[
\| f - P_{\varphi} f \| \leq \| f - P_{\varphi \perp \psi} f \| \leq (\cos \theta_{\psi,\varphi})^{-1} \| f - P_{\varphi} f \|,
\]

where \( \theta_{\psi,\varphi} \) is a measure of the “maximum angle” between the two spaces, as computed from their spectral coherence [Unser and Aldroubi, 1994].
We next examine how the generalized sampling algorithms can be applied to the problems of signal translation and scaling. These operations are obviously crucial in processing images.

5.1 Translation of discretized signals

The common practice for translating a discrete signal by an offset $h$ is to sample the translated reconstruction $\tilde{f}(\cdot - h) = \tau_h \ast \tilde{f}$ with no prefiltering (i.e., with $\psi = \delta$):

$$c_h = J \tau_h \ast \tilde{f} = \tau_h \ast c = c \ast (\tau_h \ast \varphi). \quad (5.1)$$

The result is the discrete convolution between the coefficient array and the sampled, translated basis $\varphi$.

**Translation using generalized sampling**  When the reconstruction kernel includes a digital filter $r$, equation (5.1) does not directly produce the desired coefficient array. Instead, we must apply the filter:

$$c_h = c \ast (\tau_h \ast \varphi) \ast r. \quad (5.2)$$
Translation and scaling

The new coefficient array \( c_h \) is then ready for reconstruction with \( \varphi \) (and further processing).

To complete the generalization, we must add a prefiltering stage to equation \( 5.2 \). Thanks to our notation, this can be done with simple algebraic manipulations:

\[
\begin{align*}
    c_h &= \left[ (\tau_h * \hat{f}) * (p * \psi) \right] * r \\
    &= \left[ \tau_h * c * \varphi * \psi \right] * p^\varphi * r \\
    &= c * \left[ \tau_h * a_{\varphi, \psi} \right] * p^\varphi * r \\
    &= c * q \quad \text{with} \quad q = \left[ \tau_h * a_{\varphi, \psi} \right] * p^\varphi * r.
\end{align*}
\]  

(5.3) (5.4) (5.5) (5.6)

To represent naïve translation within this generalized sampling framework, simply eliminate prefiltering by setting \( \psi = \delta \) and \( p = r = \delta \) in (5.6). The result is (5.1). The generalized sampling method, however, can also express more sophisticated strategies.

**Least-squares translation** Unlike the space of band-limited functions \( V_{\text{sinc}} \), the translation \( \tau_h * \hat{f} \) of a function \( \hat{f} \in V_\varphi \) does not in general belong to \( V_\varphi \). Unser et al. [1995b] therefore explore forming its orthogonal projection \( P_\varphi(\tau_h * \hat{f}) \) into \( V_\varphi \) and then sampling it. This is accomplished by setting \( p * \psi = \hat{\varphi} = \varphi * \left[ a_{\varphi} \right]^{-1} \) and \( r = \delta \) in (5.6):

\[
c_h = c * \left[ \tau_h * a_{\varphi} \right] * \left[ a_{\varphi} \right]^{-1} \quad \text{so that} \quad \left[ P_\varphi(\tau_h * \hat{f}) \right] = c_h * \left[ \varphi \right].
\]  

(5.7)

They further observe that (5.7) may be rewritten to resemble (5.1):

\[
c_h = c * \left[ \tau_h * (a_{\varphi})_{\text{int}} \right].
\]  

(5.8)

This means that least-squares translation in space \( V_\varphi \) is similar to naïve translation in the space \( V_{a_{\varphi}} \). For example, if \( \varphi = \beta^n \) is the B-spline of degree \( n \) (and order \( L = n + 1 \)), then \( a_{\varphi} = \beta^{2n+1} \) is the B-spline of degree \( 2n + 1 \) (and order \( L = 2n + 2 \)). Therefore, in this case, least-squares translation is equivalent to performing the naïve translation in a space that has twice the approximation order.

Note that in all of the translation algorithms, the reconstruction and pre-filter kernels are sampled into the digital filter, so computing the translated sequences only involves a discrete convolution.
5.2 Scaling of discretized signals

Scaling a discrete sequence is a more difficult operation than translation because in the general case it cannot be computed as a single convolution. The overall process is illustrated in figure 5.1.

We use the shorthand notation \( f_s \overset{\text{def}}{=} f(\cdot/s) \) to denote \( f \) after a uniform scale by factor \( s \). Using explicit sampling rates as in (3.4) and (3.7), we first note a few convenient relations:

\[
(f * g)_s = \frac{1}{s} f_s * g_s, \quad (5.9)
\]
\[
(c * \varphi)_s = c * \varphi_s, \quad (5.10)
\]
\[
[f_s] = [f] \frac{1}{s}, \quad \text{and} \quad (5.11)
\]
\[
f * \delta_s = sf. \quad (5.12)
\]

In scaling a discrete sequence, we distinguish between magnification (upsampling, \( s > 1 \)) and minification (downsampling, \( s < 1 \)). These cases are treated independently.

When magnifying, the prefilter becomes redundant in the presence of a good reconstruction filter. To see this, recall that a good reconstruction filter has a cut-off frequency \( \approx \frac{1}{2} \text{cycles per input sample} \), whereas a good prefilter has cut-off at \( \approx \frac{1}{2} \text{cycles per output sample} \). Since the scaling operation is such that the output sampling rate is \( s \) times the input sampling rate, the reconstruction filter’s cut-off frequency is \( s \) times lower than the prefilter’s. Therefore, common practice is to sample the scaled reconstruction without
any prefiltering (i.e., \( \psi = \delta \) when \( s \geq 1 \)):

\[
\left[ \hat{f}_s \ast \delta^s \right] = s\left[ (\hat{f} \ast \delta^s)_{s} \right] \\
= s\left[ \hat{f} \ast \delta^s_{\frac{1}{s}} \right]_{s} \\
= \left[ \hat{f} \right]_{s} \\
= \left[ c \ast \varphi \right]_{s}.
\]

(5.13) (5.14) (5.15) (5.16)

See function \texttt{upsample} in lines [202-220] of appendix [A] for source-code implementing this algorithm.

Conversely, when minifying, it is the effect of the reconstruction filter that is hidden by the prefilter. Accordingly, common practice is to prefilter the discrete signal without any reconstruction (i.e., \( \varphi = \delta \) when \( s \leq 1 \)):

\[
\left[ \hat{f}_s \ast \psi^s \right] = \left[ (c \ast \delta)_{s} \ast \psi^s \right] \\
= \left[ c \ast s \delta^s \ast \psi^s \right] \\
= s\left[ c \ast \psi^s \right] \\
= s\left[ (c \ast \psi^s_{\frac{1}{s}})_{s} \right] \\
= s\left[ c \ast \psi^s_{\frac{1}{s}} \right]_{s} \\
= \left[ c \ast (s \psi^s_{\frac{1}{s}}) \right]_{s}.
\]

(5.17) (5.18) (5.19) (5.20) (5.21) (5.22)

Note that the factor \( s \) within \( s \psi^s_{\frac{1}{s}} \) in (5.22) acts to preserve the integral of the kernel \( \psi \) as it is scaled. In general, if \( s \) is not an integer, \( s \psi^s_{\frac{1}{s}} \) does not satisfy a partition of unity even when \( \psi \) does. It is therefore common practice to add a normalization step to the sampling process. To do so, we accumulate the weights assigned to each entry in \( c \), and divide the resulting sampled value by this number. The normalization ensures the resampled sequence has the same average level as the input. See function \texttt{downsample} in lines [222-251] of appendix [A] for the corresponding source-code.

**Scaling using generalized sampling** As we did for translation, we now derive an efficient algorithm for finding the coefficient array \( c_s \) of a
scaled signal, but employing generalized reconstruction and prefiltering:

\[ c_s = \left[ \tilde{f}_s * (p * \psi^v) \right] * r \]  
(5.23)

\[ = \left[ \tilde{f}_s * \psi^v \right] * p^v * r \]  
(5.24)

\[ = \left[ c * s \varphi_s * \psi^v \right] * p^v * r \]  
(5.25)

\[ = \left[ c * (s \varphi * \psi^v) / s \right] * p^v * r \]  
(5.26)

\[ = s \left[ c * \varphi * \psi^v \right] / s * p^v * r \]  
(5.27)

\[ = s \left[ c * a_{\varphi,\psi^v} \right] / s * p^v * r. \]  
(5.28)

The traditional algorithms (5.16) and (5.22), are special cases of (5.28). Note that the mixed convolution and the sampling operation use different spacings in (5.16), (5.22), and (5.28), and relation (3.17) does not apply. Therefore, we may have to evaluate the cross-correlation term \( a_{\varphi,\psi^v} \) at the arbitrary positions \( i/s - j \), with \( i,j \in \mathbb{Z} \). This is inconvenient, because any closed-form expression for \( a_{\varphi,\psi^v} \) is specific to a given \( s \), which may only be known at runtime. Nevertheless, since \( a_{\varphi,\psi^v} \) has compact support, we can compute all required values once and reuse them for all rows and columns in the image. Furthermore, when \( s \) is rational, the values repeat and we can compute a single cycle. Finally, when \( 1/s \) is an integer, we can simply compute \( c * [a_{\varphi,\psi^v}] \) and decimate the results by \( 1/s \) or, better yet, compute only the elements in the discrete convolution that remain after decimation.

**Repeated integration** The computation required to evaluate a single sample from \( c * \psi^v / s \) in (5.22) or from \( c * a_{\varphi,\psi^v} / s \) in (5.28) increases proportionally to \( 1/s \) (i.e., with the kernel support). Fortunately, when downsampling an entire image, the increased computation per sample is cancelled by the corresponding reduction in the total number of samples required.

Nevertheless, many applications (such as texture filtering) demand repeated access to individual samples. Heckbert [1986] introduces a repeated integration strategy (based on integrated arrays [Crow, 1984, Perlin, 1985]) that enables this sampling in constant time for piecewise polynomial kernels. Furthermore, the method allows us to avoid computing the value of scaled kernels altogether. As we have seen, this is particularly convenient when dealing with \( a_{\varphi,\psi^v} \).
Translation and scaling

Heckbert considers the case of a B-spline analysis filter $\psi = \beta^n$, and sets $\varphi = \delta$ since reconstruction does not play a significant role in minification. Muñoz et al. [2001] reach the same algorithm, while including reconstruction with $\varphi = \beta^n$ and a digital filtering stage. In our notation, (5.28) becomes:

$$c_s = s \left[ c * \varphi * (\beta^n)^{\gamma} \right]_\frac{1}{2} * p^r * r$$

(5.29)

$$c_s = s \left[ c * \varphi * \beta^n \right]_\frac{1}{2} * p^r * r$$

(5.30)

Before delving into the derivation, we first provide high-level intuition. Piecewise polynomial functions can be expressed as linear combinations of shifted one-sided power functions. The expression for the B-spline $\beta^n$ can be seen in (3.34) and only involves $u^* \left( (n+1) \right)$. As is apparent from (3.35), scaling a one-sided power function is equivalent to a multiplication by a constant factor:

$$\left( u^* \left( (n+1) \right) \right)_\frac{1}{2} = s^\mu u^* \left( (n+1) \right).$$

(5.31)

Finally, sampling a mixed convolution with a one-sided power function can be performed in constant time with a combination of precomputed running sums (i.e., higher-order summed-area tables). In the case of B-splines, it turns out that only the $(n+1)^{th}$-order sum is required.

Consider the expression for the scaled B-spline function $\beta^n$ from (3.34):

$$\beta^n = \left( \Delta^* \left( (n+1) \right) * u^* \left( (n+1) \right) * \tau_{(n+1)/2} \right)_\frac{1}{2}$$

(5.32)

$$\beta^n = s \left( \Delta^* \left( (n+1) \right) * u^* \left( (n+1) \right) \right)_\frac{1}{2} \left( \tau_{(n+1)/2} \right)_\frac{1}{2}$$

(5.33)

$$\beta^n = s \Delta^* \left( (n+1) \right) _\frac{1}{2} \left( u^* \left( (n+1) \right) \right)_\frac{1}{2} \left( \tau_{(n+1)/2} \right)_\frac{1}{2}$$

(5.34)

$$\beta^n = s^\mu \Delta^* \left( (n+1) \right) _\frac{1}{2} u^* \left( (n+1) \right) \tau_{(n+1)/(2s)}$$

(5.35)

$$\beta^n = s^\mu \Delta^* \left( (n+1) \right) _\frac{1}{2} u^* \left( (n+1) \right) \tau_{(n+1)/(2s)}.$$  

(5.36)

On the right-hand side of the mixed convolution with spacing $\frac{1}{2}$ in (5.36), we can see almost all terms in the definition of the unscaled B-spline

$$\beta^n = \Delta^* \left( (n+1) \right) * u^* \left( (n+1) \right) * \tau_{(n+1)/2}.$$  

(5.37)

To obtain the missing terms without changing results, convolve (5.36) on the right-hand side with:

$$\delta = \tau_{(n+1)/2} * \tau_{(n+1)/2} \quad \text{and} \quad \delta = u^* \left( (n+1) \right) \Delta^* \left( (n+1) \right).$$  

(5.38)

\footnote{We must be careful because the mixed convolutions involve different spacings.}
5.2. Scaling of discretized signals

Thus,
\[ \beta_n = s^n \Delta^{*n+1} * \frac{1}{2} \beta^n * u^{*n+1} * \tau_s(n+1)/(2s). \] (5.39)

Substituting into (5.30), we finally get
\[ c_s = s^{n+1} \left[ c * (\Delta^{*n+1} * \varphi * \beta^n * \tau_h * u^{*n+1}) \right] \frac{1}{\tau} * p' * r \] (5.40)
\[ = s^{n+1} \left[ \Delta^{*n+1} * \frac{1}{\tau} (c * \varphi * \beta^n * \tau_h * u^{*n+1}) \right] \frac{1}{\tau} * p' * r \] (5.41)
\[ = s^{n+1} \left[ c * u^{*n+1} * \varphi * \beta^n * \tau_h \right] \frac{1}{\tau} * \Delta^{*n+1} * p' * r \] (5.42)
\[ = [d * \phi] \frac{1}{\tau} * q, \] (5.43)

where
\[ d = c * u^{*n+1}, \] (5.44)
\[ \phi = s^{n+1} \varphi * \beta^n * \tau_h, \quad \text{with} \quad h = -\frac{(n+1)(1-s)}{2s}, \quad \text{and}, \] (5.45)
\[ q = \Delta^{*n+1} * p' * r. \] (5.46)

To summarise, the steps of the algorithm are as follows:

1. Obtain a new sequence \( d \) by applying \( u^{*n+1} \) to the input array \( c \) (thus at input resolution). This is equivalent to \( n + 1 \) successive running sums, which can also be implemented (more efficiently) as a single recursive filter of order \( n + 1 \);

2. Reconstruct with \( \varphi * \beta^n \) shifted by \( h \), multiply by \( s^{n+1} \), and sample with spacing \( \frac{1}{\tau} \). The cost of computing each output sample is independent of \( s \), requiring \( n+1 \) coefficient array accesses when \( \varphi = \delta \) and \( m+n+2 \) when \( \varphi = \beta^n \);

3. Apply the digital filter \( q \) to the output (thus at output resolution). This combines a direct convolution with filter \( \Delta^{*n+1} \), which has support \( n + 2 \), and with filters \( p' \) and \( r \).

As noted by [Heckbert 1986] and [Muñoz et al. 2001], the repeated integration framework can be extended beyond B-splines to arbitrary piecewise polynomial kernels such as the Catmull-Rom filter. However, this generalization may require the computation of multiple coefficient arrays \( d_i = u^{*i} * c \).
Translation and scaling

Least-squares scaling

Unser et al. [1995a] explore the use of orthogonal projection for scaling. This is accomplished by setting $\psi^* p = \hat{\varphi} = \varphi [a_{\varphi}]^{-1}$ and $r = \delta$ in (5.28):

$$c_s = s\left[c * a_{\varphi,\varphi}^1\right] \ast [a_{\varphi}]^{-1} \text{ so that } P_{\varphi \hat{f}_s} = c_s * [\varphi]. \quad (5.47)$$

Restricting $\varphi$ to the family of B-splines, Unser et al. [1995a] provide explicit formulas for the cross-correlation function in the piecewise-constant $\varphi = \beta^0$ and piecewise-linear $\varphi = \beta^1$ cases, and note that for higher orders the cross-correlation function $a_{\beta^m,\beta^1}^n$ quickly converges to a Gaussian (by the Central Limit Theorem).

In contrast, Lee et al. [1998] approach the problem from the consistent sampling perspective. The idea is to reconstruct with $\varphi$ but prefilter with a kernel $\psi$ that simplifies the computations. Lee et al. describe how to efficiently implement the case where $\varphi = \beta^n$ and $\psi = \beta^0$. Recall from (4.41) in section 4.3 that when reconstructing with $\varphi$ and prefiltering with $\psi$, the digital correction filter that achieves oblique projection is simply $q = [\varphi * \psi^\perp]^{-1}$.

Applying this correction to (5.28) and setting $p = r = \delta$ we obtain:

$$c_s = s\left[c * a_{\varphi,\psi}^1\right] \ast [a_{\varphi,\psi}]^{-1} \text{ so that } P_{\varphi \perp \hat{f}_s} = c_s * [\varphi]. \quad (5.48)$$

Muñoz et al. [2001] apply the repeated integration strategy to the problem of oblique projection of scaled signals, essentially combining the work of Lee et al. [1998] and of Heckbert [1986]. To do so, simply set $\varphi = \beta^m$, $p = \delta$, and $r = [a_{\varphi,\beta^n}]^{-1} = [\beta^{m+n+1}]^{-1}$ in (5.42) to reach:

$$c_s = s_{n+1} \left[c * u^{(n+1)} * \beta^{m+n+1} \ast \tau_h\right] \ast [\beta^{m+n+1}]^{-1}.$$

As Muñoz et al. [2001] note, the extension of the repeated integration method to linear combinations of B-splines and their derivatives allows the method to support the entire MOMS family [Blu et al., 2001].

Nearest-neighbor and linear filtering

We can now give precise definitions for the terms “linear” and “nearest-neighbor” that are widely used in relation to magnification and minification. Using (5.28) as a starting point, we first eliminate the digital filtering by setting $p = q = \delta$ since these are traditional algorithms. Intuitively, nearest-neighbor magnification means directly
5.2. Scaling of discretized signals

sampling the box reconstruction ($\varphi = \beta^0, \psi = \delta$):

$$c_s = [c \ast \beta^0]_\frac{1}{s}. \quad (5.49)$$

Conversely, linear magnification means directly sampling the hat reconstruction ($\varphi = \beta^1, \psi = \delta$):

$$c_s = [c \ast \beta^1]_\frac{1}{s}. \quad (5.50)$$

In a perhaps counter-intuitive fashion, linear minification means directly pre-filtering with the box kernel ($\varphi = \delta, \psi = \beta^0$):

$$c_s = s[c \ast \beta^0]_\frac{1}{s}. \quad (5.51)$$

This breaks the analogy to linear magnification, which uses the hat filter. Finally, nearest-neighbor minification uses the exact same equation as nearest-neighbor magnification. But since $s \leq 1$ for minification, the absence of a prefilter potentially leads to severe aliasing.

Pyramid specialization (Mipmaps)  
Equation (5.28) describes a family of downscaling algorithms parametrized by the choices of generalized analysis and reconstruction kernels, and by the scaling factor $s$. Here we describe a specialization that is useful in the generation of dyadic pyramids or mipmaps (i.e., $s = \frac{1}{2}$).

As discussed later in section [10], the cardinal cubic B-spline, i.e. $\psi \ast p = (\beta^3)_{\text{int}}$, is an excellent prefilter. Using the traditional approach of reconstructing with the impulse $\varphi \ast r = \delta$ and substituting in (5.28), we reach the following algorithm for the recursive generation of level $i$ in an image pyramid, starting from an input image $f_1$:

$$f_i = \frac{1}{2} [\beta^3]^{-1} * [f_{i-1} \ast [\beta^3]^2]_\frac{1}{2}. \quad (5.52)$$

The direct convolution is with kernel $[\beta^3]$, which has support 7. In the case of a dual-grid mipmap structure (see section [8]), the direct convolution kernel is replaced by $[\beta^3 \ast \tau_{1/2}]$, which has support 8. Only the elements surviving decimation must be computed. These elements undergo inverse convolution with kernel $[\beta^3]$, which is symmetric with support 3, as per section [4.2].

Exploiting separability and normalization, the full 2D process requires only 35
floating-point operations per output pixel, per channel (only 27 with multiply-add instructions (MADs)). In comparison, Catmull-Rom kernel requires 30 (22 with MADs), and a Lanczos windowed sinc ($W = 6$) requires 49 (34 with MADs).
Approximation of derivatives

In computer graphics and visualization, it is often desired to reconstruct not only the function but also its derivatives. For example, first derivatives contribute tangent vectors for surface shading, and second derivatives allow edge detection with subpixel accuracy.

The theory presented so far applies equally well to the $n$th derivative $f^{(n)}$ of a given function $f$, so long as we have access to $f^{(n)}$. In that case, we can subject it to the generalized sampling pipeline and find its approximation in the order-$L$ space $V_\varphi$ of our choice:

$$\hat{f}^{(n)} = [f^{(n)} \ast \psi^{(l)}] \ast q \ast \varphi,$$

in the same way we would have obtained an approximation to $f$:

$$\hat{f} = [f \ast \psi^{(l)}] \ast q \ast \varphi. \quad (6.2)$$

However, our access is often restricted to $\hat{f}$. Given only the coefficient array $[f \ast \psi^{(l)}]$, we wish to obtain an approximation to $f^{(n)}$. The most obvious approach is to differentiate $\hat{f}$, which is equivalent to reconstructing with the derivative $\varphi^{(n)}$ of the original kernel:

$$(\hat{f})^{(n)} = ([f \ast \psi^{(l)}] \ast q \ast \varphi)^{(n)} = [f \ast \psi^{(l)}] \ast q \ast \varphi^{(n)}. \quad (6.3)$$
Unfortunately, reasoning about the approximation order of this scheme is not easy. We may even fail to obtain an approximation to $f^{(n)}$ as we reduce the sample spacing. For one thing, the approximation order cannot be higher than the order of the new approximation space $V_{\varphi^{(n)}}$ where $(\hat{f})^{(n)}$ lives.

Some important tools for understanding and solving this problem are presented in the work of Condat and Möller [2011]. The insight is to use properties (3.29) and (3.34):

$$J f^* \psi^\lor K = q f^* \psi^\lor u^* \nu^* f^*$$

$$= q f^* \psi^\lor y^*$$

$$= q f^* \beta^{n-1} * \tau_{n/2} * \psi_{\lor}$$

and therefore

$$[f * \psi^\lor] * \Delta^{*n} = [f^{(n)} * \beta^{n-1} * \tau_{n/2} * \psi^\lor]$$

$$= [f^{(n)} * (\beta^{n-1} * \tau_{-n/2} * \psi)^\lor]$$

$$= [f^{(n)} * \xi_{\lor}], \text{ with } \xi = \beta^{n-1} * \tau_{-n/2} * \psi. \ (6.10)$$

In other words, starting from the samples of $f$ filtered with $\psi$, and after applying the discrete derivative $\Delta^{*n}$, we obtain samples of $f^{(n)}$, but these samples come filtered with $\xi$ instead. We can now apply our standard tools to reason about the asymptotic behavior of the approximation

$$\tilde{f}^{(n)} = [f^{(n)} * \xi_{\lor}] * q * \varphi$$

$$= [f * \psi^\lor] * \Delta^{*n} * q * \varphi. \ (6.11)$$

For example, to guarantee that we achieve the optimal approximation order in $V_{\varphi}$, we can use oblique projection and set $q = [a_{\varphi,\xi}]^{-1}$. Another remarkable consequence from this analysis is that we can obtain from samples of $f$ the exact orthogonal projection of $f^{(n)}$ in space $V_{\beta^{n-1}(-, + \frac{n}{2})}$, since in that case the oblique projection reduces to orthogonal projection.

**Concrete examples** Consider the typical case in which we are provided with samples from $f$ and we assume $\psi = \delta$. As an example, we derive a 4th-order approximation to the second derivative $f^{(2)}$ of $f$ in the space $V_{\beta^3}$ of
cubic B-splines. To do so, we simply use oblique projection by setting \( \psi = \delta \) and \( \varphi = \beta^3 \) in (6.11), from which \( \xi = \beta^1 \ast \tau_1 \):

\[
\tilde{f}^{(2)} = \left[ f^{(2)} \ast \xi^* \right] \ast \left[ \varphi \ast \xi^* \right]^{-1} \ast \varphi \tag{6.12}
\]

\[
= [f] \ast \Delta^{-2} \ast \left[ \beta^3 \ast \beta^1 \ast \tau_1 \right]^{-1} \ast \beta^3 \tag{6.13}
\]

\[
= [f] \ast [1, -2, 1] \ast \left[ \beta^5 \ast \tau_1 \right]^{-1} \ast \beta^3 \tag{6.14}
\]

\[
= [f] \ast [1, -2, 1] \ast \left[ \beta^5 \right]^{-1} \ast \beta^3. \tag{6.15}
\]

Incidentally, this is equivalent to taking the second derivative of the interpolation of \( f \) in the space of quintic B-splines, since

\[
(\beta^5)^{(2)} = (\beta^1 \ast \beta^3)^{(2)} \tag{6.16}
\]

\[
= (\Delta^{-2} \ast u^{-2} \ast \tau_{-2/2})^{(2)} \ast \beta^3 \tag{6.17}
\]

\[
= \Delta^{-2} \ast \tau_{-1} \ast \beta^3 \tag{6.18}
\]

\[
= [1, -2, 1] \ast \beta^3. \tag{6.19}
\]

Conversely, the naïve idea of differentiating the interpolation of \( f \) in space \( V_{\beta^3} \) is equivalent to performing the oblique projection of \( f^{(2)} \) into space \( V_{\beta^1} \); it is therefore only a second-order approximation strategy.

Similarly, for the 4th-order approximation to the first derivative, we have:

\[
\tilde{f}' = [f] \ast \Delta \ast \left[ \beta^4 \ast \tau_{1/2} \right]^{-1} \ast \beta^3. \tag{6.20}
\]

Note the half-pixel shift that breaks the analogy with the derivative of the interpolation in space \( V_{\beta^4} \), which would result in:

\[
\tilde{f}' = [f] \ast \Delta \ast \left[ \beta^4 \right]^{-1} \ast \beta^3 \ast \tau_{-1/2}. \tag{6.21}
\]

The latter is, of course, what we would obtain by performing the oblique projection of \( f' \) in the space of half-shifted cubic B-splines \( V_{\beta^3(\cdot + \frac{1}{2})} \), and therefore is also a 4th-order approximation scheme.

Often, we must reconstruct estimates for both \( f \) and \( f^{(n)} \). The concrete methods we described so far assume we have independent coefficient arrays for \( \tilde{f} \) and \( \tilde{f}^{(n)} \). This is because the digital filters must be precomputed, since they are inverse discrete convolutions. In volume-rendering applications that require gradients, keeping 4 versions of the coefficient volume (for \( f \) and its 3 partial derivatives) can be prohibitive in terms of memory consumption. Fortunately, we are not required to use oblique projection to guarantee an optimal
approximation order to $f^{(n)}$. (See section 9 for details.) In recent follow-up work, Alim et al. [2010] used this additional freedom to derive FIR filters that can be applied locally, on demand, on the interpolating coefficients for $f$. This allows sharing the same coefficient vector for optimal-order approximations to $f$ and its partial derivatives.
We previously explored how images can be downscaled using a generalized prefilter, expressed as a mixed convolution of a compact kernel $\psi$ and a digital filter $p$ (section 5.2). This strategy attains efficiency due to the compactness of kernel $\psi$ yet reduces aliasing and maintains sharpness due to the improved frequency response of the effective kernel $\psi \ast p$.

We now apply this same generalized prefiltering strategy to the case where the input function $f$ is defined procedurally rather than sampled. This is particularly relevant to computer graphics because procedural definitions abound in the rendering process:

- Vector representations define shapes with intricate boundaries;
- Procedural texturing assigns material properties to surfaces;
- Procedural lighting defines local illumination over the surface;
- Procedural shading evaluates color from materials and lighting;
- Perspective projection maps scenes surfaces to the rendered image.

An important challenge in rendering is that any of these procedures may introduce high frequencies into the rendered image $f$. Troublesome cases
include sharp outlines, noise textures, shadow discontinuities, specular highlights, surface silhouettes, and perspective distortions near vanishing points.

The general approach to overcome this problem of discontinuities is to prefilter the function $f$, typically by supersampling. We evaluate $f$ at several positions per pixel and integrate these samples using a prefilter kernel. The resulting estimate at each pixel suffers from aliasing and noise. Aliasing is attenuated by improving the frequency response of the prefilter, and noise is attenuated by reducing the variance of the supersampling estimate. We next analyze these characteristics in the generalized sampling pipeline. In particular, we investigate whether it is advantageous to reuse samples across pixels when evaluating prefilter kernels.

Computing the prefiltered sample at pixel $k$ from signal $f$ entails evaluating the integral:

$$ b_k = (f * \psi)(k) = \int_{-\infty}^{\infty} f(x) \psi(x - k) \, dx. \quad (7.1) $$

When exact evaluation is impractical, the integral is approximated by Monte Carlo integration (i.e., by supersampling). The idea is to rewrite the integral as an expectation, and to approximate it with a sample mean. Although changing the sample distribution can reduce the variance of the estimator (see quasi-Monte Carlo, importance sampling), or even reshape its spectral properties [Dippé and Wold, 1985], here our interest is analyzing the effect of the generalized prefilter $\psi * p$ on the variance of the estimator.

**Sharing samples**  To share samples between pixels, we start by tiling the real line with the support of uniformly distributed random variables. To that end, let $X_i \sim U(-.5,.5) + i - o_W$, where $o_W$ takes value .5 when the support width $W$ of $\psi$ is even, and 0 otherwise. Using $X_i$, we can rewrite (7.1) as a sum of expectations:

$$ b_k = \sum_{i \in \mathbb{Z}} E_{X_i}(f(\cdot) \psi(\cdot - k)), \quad (7.2) $$

since $E_{X_i}(g) = \int g(x) p_{X_i}(x) \, dx$ and the uniform probability density function $p_{X_i} = 1$. To make the analysis practical, we assume that $f$ is stationary and $f(X_i)$ is uncorrelated to $X_i$. This is reasonable if the frequency content of $f$ is much higher than the supersampling rate. Therefore, $E_{X_i}(f) = E_{X_j}(f)$ for all $i$ and $j$, and we denote the common value by $E_X(f)$. 

The unbiased Monte Carlo estimator for $b_k$ is:

$$
\hat{b}_k = \frac{1}{m} \sum_{j=1}^{m} \sum_{i \in \mathbb{Z}} f(x_{i,j}) \psi(x_{i,j} - k),
$$

(7.3)

where $m$ variates $x_{i,j}$ are drawn from each $X_i$. Because samples are shared between $\hat{b}_k$ and $\hat{b}_{k+w}$ whenever $|w| < W$, the estimators $\hat{b}_k$ and $\hat{b}_{k+w}$ are not independent random variables. The covariance is:

$$
\text{Cov}(\hat{b}_k, \hat{b}_{k-w}) = \frac{1}{m} E_X(f^2) \sum_i E_{X_i}(\psi(\cdot) \psi(\cdot + w)) - \frac{1}{m} E_X^2(f) \sum_i E_{X_i}(\psi(\cdot)) E_{X_i}(\psi(\cdot + w)),
$$

(7.4)

and the variance (i.e. $w = 0$) simplifies to:

$$
\text{Var}(\hat{b}_k) = \frac{1}{m} E_X(f^2) \sum_i E_{X_i}(\psi^2) - \frac{1}{m} E_X^2(f) \sum_i E_{X_i}^2(\psi).
$$

(7.5)

In other words, the kernels that reduce the variance the most are those with small $\sum_i E_{X_i}(\psi^2)$ and large $\sum_i E_{X_i}^2(\psi)$.

As for the digital filter $p$, it acts by computing a weighted sum of $\hat{b}_k$:

$$
c = p \ast b \Rightarrow \hat{c}_k = \sum_{i \in \mathbb{Z}} p_i \hat{b}_{k-i}.
$$

(7.6)

Keeping in mind that the $b_k$ are not independent,

$$
\text{Var}(\hat{c}_k) = \sum_{w \in \mathbb{Z}} \sum_{i \in \mathbb{Z}} p_i p_{i+w} \text{Cov}(\hat{b}_{k-i}, \hat{b}_{k-i-w})
$$

(7.7)

$$
= \sum_{w=-W+1}^{W-1} (a_p)_w \text{Cov}(\hat{b}_k, \hat{b}_{k-w}).
$$

(7.8)

Only $2W - 1$ elements from sequence $a_p = p \ast p'$ are associated to nonzero covariances. Furthermore, since $a_p$ is symmetric, only $W$ of them are distinct. These can be computed exactly via integrals of the DTFT of $p$ (using Parseval’s theorem and the time-shift property). Substituting (7.4) and (7.5) into (7.8), and collecting together the terms multiplying $E_X(f^2)$ and $E_X^2(f)$ into coefficients $S_{\psi,p}$ and $s_{\psi,p}$, respectively, we reach our final expression for the variance:

$$
\text{Var}(\hat{c}_k) = \frac{1}{m} (S_{\psi,p} E_X(f^2) - s_{\psi,p} E_X^2(f)).
$$

(7.9)
Table 7.1: Analysis of variance when supersampling with various prefilters. Constants \(S_{\psi,p}\), \(s_{\psi,p}\), \(N_{\psi,p}\), and \(n_{\psi,p}\), are the factors from (7.9) and (7.13). Column \(\sigma_U\) gives the standard deviation of estimator \(\hat{c}_k\) using \(m = 1\) sample per unit pixel area and signal \(f \sim U(0,1)\) uniformly distributed. (\(\beta_{\text{int}}^N\) the cardinal B-splines; \(K\) the Catmull-Rom filter; \(M\) the Mitchell-Netravali filter.)

<table>
<thead>
<tr>
<th>Prefilter</th>
<th>Sharing samples</th>
<th>No sample sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S_{\psi,p})</td>
<td>(S_{\psi,p})</td>
</tr>
<tr>
<td>(\beta_{\text{int}}^5)</td>
<td>.92</td>
<td>.84</td>
</tr>
<tr>
<td>(\beta_{\text{int}}^3)</td>
<td>.87</td>
<td>.76</td>
</tr>
<tr>
<td>Lanczos</td>
<td>.89</td>
<td>.79</td>
</tr>
<tr>
<td>(K)</td>
<td>.81</td>
<td>.66</td>
</tr>
<tr>
<td>(M)</td>
<td>.68</td>
<td>.46</td>
</tr>
<tr>
<td>(\beta^0)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**No sample sharing** The analysis is simpler when no sample-sharing is involved. We use independent variables \(Y\), uniformly distributed over the support \(\Omega\) of \(\psi\) (with area \(A_\Omega\)). Equation (7.3) simplifies to:

\[
\hat{b}_k = \frac{A_\Omega}{m} \sum_{j=1}^{m} f(y_j) \psi(y_j - k).
\]  

(7.10)

Independence also leads to simpler versions of (7.5) and (7.8):

\[
\text{Var}(\hat{b}_k) = \frac{A_\Omega^2}{m} E_Y(f^2) E_\psi^2 - \frac{A_\Omega^2}{m^2} E_Y^2(f) E_\psi^2, \quad \text{and} \quad \text{Var}(\hat{c}_k) = \|p\|^2 \text{Var}(\hat{b}_k).
\]  

(7.11)

(7.12)

Collecting the terms on \(E_Y(f^2)\) and \(E_Y^2(f)\) into coefficients \(N_{\psi,p}\) and \(n_{\psi,p}\), respectively, we reach an expression analogous to (7.9):

\[
\text{Var}(\hat{c}_k) = \frac{1}{m} (N_{\psi,p} E_Y(f^2) - n_{\psi,p} E_Y^2(f)).
\]  

(7.13)

In the second term of (7.11), note that \(E_Y(\psi) = 1 / A_\Omega\) if the kernel \(\psi\) has unit integral. Therefore this second term simplifies to \(-\frac{1}{m} E_Y^2(f)\), which is independent of the choice of kernel \(\psi\). Consequently, we get \(n_{\psi,p} \approx 1\) for all the traditional kernels, i.e. those for which the digital filter \(p\) is the identity. (The Lanczos kernel does not precisely have unit integral, so its value \(n_{\psi,p}\) differs slightly from 1.)

\(^1\)Note that \(E_Y^2(\psi) = 1\) for normalized prefilters.
Figure 7.1: A $128^2$ zonal plate, rendered with $5^2$ stratified samples per pixel spacing area. (a-f) Various prefilters show particular sharpness and aliasing-suppression properties, but lead to similar noise reduction. (g,h) Sharing of samples is crucial to prevent the digital filtering stage from boosting noise, as seen when we independently integrate each pixel with $20^2$ stratified samples (g), or even when using 400 importance-samples (h).

**Discussion** The values of constants $S_{\psi,p}$, $s_{\psi,p}$ $N_{\psi,p}$ and $n_{\psi,p}$ for a variety of filtering strategies are shown in table 7.1. As a measure of visible noise, the table also lists the standard deviation of estimators ($m = 1$) when sampling from a white-noise function.

As expected, the prefilter that gives the greatest factor in variance reduction is the box filter, but its low-pass properties are lacking (see figure 7.1b). The Mitchell-Netravali filter is noteworthy in having similar variance reduction and greatly improved response.

The most important takeaway from this analysis is that the remaining filters (in particular, the nice cardinal splines highlighted in blue) are not that much worse in terms of variance reduction, *as long as samples are shared*. The correlations due to sharing act to attenuate the effect of the digital filter $p$ over the variance, which is otherwise large (as highlighted in red in the table). In fact, sharing samples performs better then not sharing even when the number of samples integrated under each kernel support is the same in both cases (compare figure 7.1f,g), and even when using importance sampling.
(compare figure \ref{fig:7.1:h}). And of course, sharing samples is a huge benefit computationally, as it significantly reduces the number of evaluations of the input signal.
8

Practical considerations

8.1 Grid structure

Given a function $f$ defined over domain $[0, 1]$, there are several conventions for where to position $n$ uniformly spaced samples. The simplest may be the primal grid structure, which places sample $c_k$ at position $k/n$. However, this leads to asymmetry at the boundaries since the first sample is at 0 and the last sample is at $(n-1)/n \neq 1$. Samples can also be placed at positions $k/(n-1)$ but this requires non-power-of-two resolutions in an image pyramid. For these reasons, computer graphics usually prefers the dual grid structure, which places sample $c_k$ at position $(k + \frac{1}{2})/n$. We have adopted the primal structure during our derivations, but it is easy to transition between the two. Given the sample spacing $T = 1/n$, the primal-grid operations of sampling and reconstruction are:

$$c_T = \left[ f * \psi^\vee_T \right]_T \iff \hat{f} = c_T * \varphi_T,$$  \hspace{1cm} (8.1)

whereas the dual-grid counterparts are:

$$c_T = \left[ f * \psi^\vee_T \left( \cdot - \frac{T}{2} \right) \right]_T \iff \hat{f} = c_T * \varphi_T \left( \cdot - \frac{T}{2} \right).$$  \hspace{1cm} (8.2)

In a multiresolution representation, it is useful to obtain a nesting of the function spaces defined at the various resolutions, so that any reconstruction
at a coarser level can be represented exactly in the space of reconstructions at finer levels. The B-spline family of kernels offers this refinability. The main constraint is that even-degree kernels must follow a primal structure, whereas odd-degree kernels must follow a dual structure. Thus, for compatibility with the dual-grid structure, we often use B-splines of degree 3 and 5.

### 8.2 Efficient use of piecewise-polynomial kernels

To illustrate the efficient use of piecewise-polynomial kernels, we examine the common operations of sequence magnification and minification discussed in section 5.2. We start with an intuitive implementation.

The task in magnification is to obtain the sequence \( c_s = \left[ c * \varphi \right]_s \) expressed in (5.16). Assuming a reconstruction kernel \( \varphi \) with support \( W \):

\[
(c_s)_j = \sum_{i \in \mathbb{Z}} c_i \varphi(j/s - i) = \sum_{i = \ell_j}^{r_j} c_i \varphi(j/s - i), \quad \text{with} \quad \ell_j = \left\lceil j/s - W/2 \right\rceil \quad \text{and} \quad r_j = \left\lfloor (j/s + W/2) \right\rfloor.
\]

This is the intuitive algorithm implemented by the upsample function in appendix A, lines 202–220 (except for using the dual grid structure).

The minification formula starts from \( c_s = \left[ c * (s \psi)^\top \right]_\frac{1}{s} \) as in (5.21). Assuming this time an analysis filter \( \psi \) with support \( W \):

\[
(c_s)_j = s \sum_{i \in \mathbb{Z}} c_i (s \psi)_\frac{1}{s}(j/s - i) = s \sum_{i = \ell_j}^{r_j} c_i (s \psi'(j - is)), \quad \text{with} \quad \ell_j = \left\lceil j/s - W/(2s) \right\rceil \quad \text{and} \quad r_j = \left\lfloor j/s + W/(2s) \right\rfloor.
\]

This is the intuitive algorithm implemented by the downsample function in appendix A, lines 222–251 (again except for the dual grid structure).

Equations (8.3) and (8.4) allow direct random-access evaluation of any sample \((c_s)_j\). While this freedom is useful in a variety of applications, it has its costs. First, because the kernels are piecewise polynomial, evaluating them at arbitrary offsets \( \varphi(j/s - i) \) or \( \psi'(j - is) \) requires branching on the correct polynomial piece for each loop iteration \( i \). Second, the algorithms often read each input sample multiple times. Instead, it is preferable to traverse the input only once to reduce expensive memory accesses and indexing computations near boundaries. When the task at hand is to magnify or minify an entire image
Efficient use of piecewise-polynomial kernels

8.2. Efficient use of piecewise-polynomial kernels

Figure 8.1: Efficient piecewise-polynomial kernel evaluation. The cubic B-spline kernel defined over the domain $x \in (-2, 2]$ (top left) is divided into its 4 polynomial pieces, each defined inside a unit domain $u \in (0, 1]$ (top right). During magnification, we keep track of the value of $u$ associated to each output sample. During minification, we keep track of the value of $u$ associated to each input sample. That way we always know which polynomial piece to use.

(a common case), it is possible to eliminate these inefficiencies, leading to performance improvements of 4–10×.

As an example, consider the top of figure 8.1 It shows the cubic B-spline kernel, with support $W = 4$, divided into its 4 polynomial pieces, $k_0, k_1, k_2, k_3$, each covering a unit interval. We set up the computation so that we always know which piece to use, as shown in the bottom of figure 8.1.

In the case of magnification, consider $b_0, b_1, b_2, b_3$ as input samples stored in a FIFO buffer. It is clear from figure 8.1 that to evaluate any output sample $f_u$ situated at $u$ along the way between $b_1$ and $b_2$, we can use the expression:

$$f_u \leftarrow b_0 k_3(u) + b_1 k_2(u) + b_2 k_1(u) + b_3 k_0(u).$$

(8.5)

By keeping these four samples in fast local variables, we can generate all output samples in the interval by incrementally adding $\frac{1}{4}$ to $u$ and evaluating the exact same expression. When $u > 1$, we advance the input by shifting a new sample into the FIFO buffer, subtract 1 from $u$, and repeat the process. This incremental algorithm implemented within function upsamp1e2 in lines 260-278 of appendix A.

The incremental minification algorithm is very similar. The difference is that now $b_0, b_1, b_2, b_3$ each represent an output sample accumulating weighted input samples. From figure 8.1, the contribution of an input sample $f_u$ situ-
ated $u$ along the way between $b_1$ and $b_2$ is simply:

$$
\begin{align*}
  b_0 &\leftarrow b_0 + f_u k_3(u), & b_1 &\leftarrow b_1 + f_u k_2(u), \\
  b_2 &\leftarrow b_2 + f_u k_1(u), & b_3 &\leftarrow b_3 + f_u k_0(u).
\end{align*}
$$

(8.6)

(8.7)

We can therefore iterate over the input, incrementally adding $s$ to $u$ and updating the partial output samples in the buffer using the expressions above. Whenever $u > 1$, we subtract 1 from $u$, advance the output by saving the value $s b_0$, shift a 0 into the FIFO buffer, and repeat the process. This incremental algorithm is implemented within function `downsample2` in lines 280–307 of appendix A.

**GPU reconstruction** Texture samplers on GPUs often have custom hardware to perform bilinear reconstruction at reduced cost. Moving to higher-quality reconstruction may incur a performance penalty if these bilinear samplers are underutilized. For instance, an implementation of the bicubic Mitchell-Netravali filter requires $4 \times 4 = 16$ texture reads, versus 1 read for bilinear filtering. As shown by [Sigg and Hadwiger, 2005], even though the bicubic B-spline basis has the same $4 \times 4$ support, the fact that it is non-negative and separable allows it to be evaluated by combining just 4 bilinear reads at appropriately computed locations. [Ruijters et al., 2008] also describe a similar trick. Depending on the GPU, our implementation of this idea runs 3–6× slower than bilinear filtering, but about 2× faster than Catmull-Rom or Mitchell-Netravali filters.

**8.3 Prefiltering, reconstruction, and color spaces**

In reconstruction, the average between two function values should be *perceived* to have the intermediate value between them. Luminance values do not have this property due to a nonlinear response in the human visual system. This nonlinear perceptual response to luminance is called *lightness*. The gamma correction in the sRGB standard approximates the conversion of luminance to lightness. Therefore, when processing visual data we recommend performing reconstruction directly in sRGB space to achieve better perceptual results.

In stark contrast, prefiltering is an area integral of radiant energy. It must be performed over linear intensities and only then can results be moved to a
8.4. Range constraints

An important consideration is that the digital filter $q$ may extend the range of values in an image. For example, a signal $f$ with range bounded to $[0, 1]$ may require a coefficient vector $c$ with values outside this range, especially if $f$ has high-frequency content. While this larger range is likely acceptable for floating-point or HDR image representations, it is a concern for common 8-bit fixed-point formats.

To find an approximation for interpolation coefficients $c$ within an 8-bit image format, we set its quantized range to a somewhat larger interval $[\ell, h]$, and solve the constrained optimization:

$$\hat{c} = \arg \min_c \|c * [\varphi] - f\|^2, \quad \text{s.t.} \quad \ell \leq c_k \leq h.$$  \hspace{1cm} (8.8)

This is a least squares problem with simple bounds constraints. Efficient direct solvers exist, e.g. the Matlab function lsqlin. Alternatively, we obtain good approximations using just five iterations of Gauss-Seidel relaxation, wherein the update of each pixel value is clamped to the bounds constraints.
Experimenting on natural images, we find that, for bicubic B-splines, setting the expanded range $[\ell, h] = [-0.5, 1.5]$ gives good results with respect to both an $L_2$ and SSIM metric. One might expect that the sampled reconstruction $\tilde{f}$ may only interpolate a given input $f$ with 7 bits of precision, because the quantized $c$ values now span twice the range. Surprisingly, this is not the case for natural images. With the bicubic B-spline basis, each pixel is reconstructed by a weighted combination of 16 coefficients of $c$. This combination effectively provides sub-quantum precision. Even with the simple constrained optimization (8.8), which is unaware of the subsequent quantization process, a reconstruction of the original image of figure 10.5 from an 8-bit-quantized vector $c$ has an rms error of 0.07% and a maximum error of 12%, with about 93% of pixel values reproduced exactly. Intuitively, the bicubic B-spline gives more precision to low frequencies.

As future work one could explore a combinatorial optimization to let the 8-bit-quantized $c$ values approximate an image with more than 8 bits of precision. Such super-precision may be possible only in low-frequency regions, but these smooth regions are precisely where precision is most perceptible.
In this section, we delve deeper into theoretical considerations on the approximation order of a complete sampling pipeline. Recall that we obtain an approximation $\tilde{f}_T$ to a signal $f$ as follows:

$$\tilde{f}_T = [f \ast (q \ast \psi)_T^*]_T \ast_T \varphi_T = [f \ast \psi]^T \ast_T (q \ast \varphi)_T.$$  \hspace{1cm} (9.1)

Any given choice of generating function $\varphi$ defines the shift-invariant space $V_{\varphi,T}$ of representable approximations:

$$V_{\varphi,T} = \{ \tilde{f}_T : \mathbb{R} \rightarrow \mathbb{R} | \tilde{f}_T = c_T \ast_T \varphi_T, \forall c_T \in l_2 \}. \hspace{1cm} (9.2)$$

The roles of the digital filter $q$ and the analysis filter $\psi$ are to select a particular approximation $\tilde{f}_T$ in this space, i.e., they select the coefficient vector $c_T$.

We say that space $V_{\varphi,T}$ has approximation order $L$ when we can find a constant $C > 0$ such that the following relation holds:

$$\|f - P_{\varphi,T}f\|_{L_2} = C T^L \|f^{(L)}\|_{L_2} \text{ as } T \rightarrow 0. \hspace{1cm} (9.3)$$

In other words, the norm of the residual between $f$ and its orthogonal projection $P_{\varphi,T}f$ into $V_{\varphi,T}$ falls to zero with order $L$ when we progressively reduce sample spacing $T$.

We will see the connection between the approximation order and the ingredients $\varphi$, $\psi$, and $q$ of the sampling pipeline.
On admissible kernels

In formal presentations, it is common to open with admissibility restrictions on the generating function \( \varphi \). That is, we must ensure that \( V_{\varphi,T} \) is a closed subspace of \( \mathcal{L}_2 \) before we can discuss approximation in this particular signal subspace. We require that the set of shifted generating functions \( \varphi(\cdot/T-k), k \in \mathbb{Z} \) form a Riesz basis of \( V_{\varphi,T} \) so that any signal in \( V_{\varphi,T} \) is uniquely determined by \( c_T \). This condition is equivalent to the orthogonal projection being well-defined, which in turn is equivalent to the requirement that convolution with the sampled auto-correlation \( [a_{\varphi}] = [\varphi * \varphi^\ast] \) of \( \varphi \) must be a bounded invertible operator from \( l_2 \) into itself. Interestingly, if \( p \) is an invertible convolution operator then \( V_{p*\varphi} = V_\varphi \) and therefore \( p*\varphi \) and \( \varphi \) are equivalent generating functions. The important observation is that these safeguards are satisfied by most generating functions used in practice [Alroubi and Unser, 1994]. A key exception is that derivatives of admissible kernels are not themselves admissible.

On approximation order

The Strang-Fix conditions [Strang and Fix, 1973] state that a space \( V_\varphi \) has approximation order \( L \) if and only if all polynomials of degree up to \( L-1 \) belong to \( V_\varphi \). Any projection operator finds the unique representation of the polynomials in \( V_\varphi \). In particular, the orthogonal projection of section 4.3. The Strang-Fix conditions are therefore equivalent to:

\[
P_{\varphi} (\cdot)^n = [(\cdot)^n * \varphi^\ast] * \varphi = (\cdot)^n, \quad n \in \{0, \ldots, L-1\}.
\] (9.4)

More generally, the oblique projection of section 4.4 also finds the representation of polynomials of degree up to \( L-1 \) in \( V_\varphi \) if they exist, so long as the analysis filter \( \psi \) is such that \( [a_{\varphi,\psi}] \) is invertible:

\[
P_{\varphi,\psi} (\cdot)^n = [(\cdot)^n * \psi^\ast][a_{\varphi,\psi}]^{-1} * \varphi = (\cdot)^n, \quad n \in \{0, \ldots, L-1\}.
\] (9.5)

A simpler formulation comes from the interpolation strategy of section 4.1. It is the projection operator that results from setting \( \psi = \delta \) above:

\[
[(\cdot)^n] * \varphi_{\text{int}} = [(\cdot)^n] * \varphi^{-1} * \varphi = (\cdot)^n, \quad n \in \{0, \ldots, L-1\}.
\] (9.6)

The conditions on approximation order describe the asymptotic behavior of the orthogonal projection. Other choices of analysis filter and digital filter necessarily lead to poorer approximations with respect to the \( \mathcal{L}_2 \) norm, but
may attain the same approximation order \( L \) as the orthogonal projection. This is the case if the approximation scheme reproduces all polynomials of degree up to \( L - 1 \) exactly [de Boor, 1989].

Any projection into a space \( V_\varphi \) of order \( L \) satisfies this property. These include interpolation, orthogonal projection, and oblique projection. Enforcing the projection property is equivalent to requiring the combination of analysis filter \( \psi \) and digital filter \( q \) to be biorthogonal to \( \varphi \):

\[
[\varphi \ast (q' \ast \psi')] = \delta. \tag{9.7}
\]

Although sufficient, this is not a necessary condition. A projection exactly reproduces all functions in \( V_\varphi \). This includes many other functions besides the polynomials of degree up to \( L - 1 \) that we care about. The necessary and sufficient condition is simply

\[
[(\cdot)^n \ast (q' \ast \psi')] = [(\cdot)^n \ast \varphi'], \quad n \in \{0, \ldots, L-1\}. \tag{9.8}
\]

This is in fact equivalent to requiring equality only at the sample placed at 0. It is enough for the effective analysis filter to have same moments as the dual up to order \( L - 1 \) [Blu and Unser, 1999b]:

\[
\langle (\cdot)^n, q' \ast \psi \rangle = \langle (\cdot)^n, \varphi \rangle, \quad n \in \{0, \ldots, L-1\}. \tag{9.9}
\]

This milder condition is called \textit{quasibiorthogonality of order} \( L \), and the resulting schemes are called \textit{quasiinterpolators of order} \( L \).

**Frequency domain characterizations** For many of the properties we discussed in this section, the frequency domain characterizations bring valuable insight. For example, the admissibility criterion on \( V_\varphi \) is equivalent to the sampled auto-correlation \( a_\varphi = [a_\varphi] \) of \( \varphi \) being a bounded, invertible discrete convolution operator. In the frequency domain

\[
b \ast a_\varphi \overset{F}{\longrightarrow} \hat{b} \hat{a}_\varphi \quad \text{and} \quad c \ast (a_\varphi)^{-1} \overset{F}{\longrightarrow} \frac{\hat{c}}{\hat{a}_\varphi}. \tag{9.10}
\]

Boundedness and invertibility of the operator are equivalent to the conditions

\[
A \leq \hat{a}_\varphi \leq B, \quad \text{for} \quad 0 < A \leq B < \infty, \tag{9.11}
\]

which are easier to verify in practice.
In more theoretical texts, we often find the sampled auto-correlation function defined directly in the frequency domain. Recall

\[ a_\varphi = [a_\varphi] = [\varphi \ast \varphi^\prime] = \mathcal{F} \cdot (\varphi \ast \varphi^\prime). \]  

(9.12)

Taking the Fourier transform,

\[ \hat{\mathcal{F}} \cdot (\hat{\varphi} \cdot (\hat{\varphi})^*) = \mathcal{F} \cdot (\hat{\varphi} \cdot (\hat{\varphi})^*) = \mathcal{F} \cdot |\hat{\varphi}|^2 = \sum_{k \in \mathbb{Z}} |\hat{\varphi}(\cdot + k)|^2, \]  

(9.13)

so that

\[ \hat{a}_\varphi = \sum_{k \in \mathbb{Z}} |\hat{\varphi}(\cdot + k)|^2. \]  

(9.14)

An equivalent formulation for (9.6) is given in [Unser et al., 1995b]:

\[ \mathcal{F} \cdot (\cdot)^n \varphi_{\text{int}}(\cdot) = \begin{cases} 1 & n = 0, \quad \text{(partition of unity)} \\ 0 & n \in \{1, \ldots, L - 1\} \end{cases}. \]  

(9.15)

Now if we recall the Fourier transform rule for derivatives

\[ (\cdot)^n f \overset{\mathcal{F}}{\to} \left(\frac{i}{2\pi}\right)^n (\hat{f})^{(n)}, \]  

(9.16)

we reach the frequency-domain formulation for (9.6):

\[ \left[ (\varphi_{\text{int}})^{(n)} \right] = \begin{cases} \delta & n = 0, \\ 0 & n \in \{1, \ldots, L - 1\} \end{cases}. \]  

(9.17)

The form in (9.17) indicates that, as the approximation order increases, the frequency response of the cardinal \( \varphi_{\text{int}} \) gets closer to the ideal low-pass filter.

The quasibiorthogonality conditions in (9.9) are also difficult to verify as stated. Recall that both the effective analysis filter and the dual are likely to have infinite support, so the resulting integrals are improper and involve infinite summations. Fortunately, the expressions in frequency domain are much simpler. If we recall that:

\[ \hat{f}(0) = \int_{-\infty}^{\infty} f(t) \, dt, \]  

(9.18)

we see that the conditions in (9.9) are equivalent to

\[ (\varphi^\prime \ast \psi)^{(n)}(0) = (\hat{\varphi})^{(n)}(0), \quad n \in \{0, \ldots, L - 1\}. \]  

(9.19)

In other words, the Maclaurin series of the Fourier transforms of the effective analysis filter and the dual must be the same up to order \( L - 1 \). The formulation in the frequency domain is again easier to verify or enforce.
Experiments and analyses

We performed experiments on about 50 kernels described in the literature. Here we present a subset of these results. More details are available in the supplemental material.

**Transient responses** Figure 10.1 shows the impulse and ramp responses for a variety of reconstruction kernels. The impulse responses for the B-spline kernels show the generator $\varphi$ as a dashed line, and the cardinal $\varphi \ast [\varphi]^{-1}$ as a solid line. The presence of negative lobes in an impulse response is often cited as a concern for possible ringing artifacts in reconstruction. However, images with good signal-to-noise ratio and proper antialiasing rarely contain impulses or even sharp step-discontinuities. Instead, we think that the ramp response is more representative of content present in natural images, and it predicts significantly reduced ringing for all kernels.

**Continuous frequency response** Another popular analysis tool for prefiltering and reconstruction kernels is the frequency response. In traditional sampling, this is simply the magnitude of the Fourier transform of $\psi$ or $\varphi$. In generalized sampling, we must account for the effect of the digital filter $q$ by multiplying-in its DTFT. Figure 10.2 shows the resulting responses for typical
Figure 10.1: Transient responses. Any reconstruction kernel containing negative lobes results in a certain amount of ringing (b–f). Mitchell and Netravali [1988] designed their kernel to minimize such problems (d), as seen in the comparison with the impulse responses of other kernels, in particular B-spline interpolation (b,c) and Lanczos (f). The ramp response, which is more characteristic of properly antialiased images, predicts significantly reduced ringing across all kernels.
Figure 10.2: Frequency responses. (a–c) The frequency response of B-spline interpolation converges towards that of the ideal low-pass filter as the degree increases. (b) $(\beta^1)_{int}$ outperforms the popular cubic kernels (d) Mitchell and Netravali [1988] and (e) Keys [1981], and its low-pass behavior is at least as good as the more expensive Lanczos kernel (f). Using the same support, $(\beta^5)_{int}$ in (c) performs significantly better.
kernels used in computer graphics. For each filter, the top graph uses a linear vertical scale and the bottom graph uses a log vertical scale (in decibels) to more clearly reveal the attenuation of unwanted high frequencies. The “ideal” frequency response corresponds to the dashed line depicted in each plot. The progression towards ideal low-pass behavior for increasing approximation order, as indicated by equation (9.17), is clearly confirmed in the plots.

**Frequency response in resampling** Figure 10.3 shows the effects of a resampling strategy on the amplitude of each frequency in the discrete input signal, as well as the phase errors introduced. The work of Parker et al. [1983] served as an inspiration for this visualization. Recall from section 5.1 that naïve translation by an offset $h$ after reconstruction with $r * \varphi$ is equivalent to discrete convolution with $r * \left[\tau_h * \varphi\right]$. Shaded areas represent the span of behaviors of the discrete kernels associated to $h \in [-0.5, 0.5]$. Thus, the frequency response can vary significantly as we vary the alignment between the input and output sample grids. It is clear that generalized kernels result in better amplitude preservation and lower phase errors. The lower phase errors may be useful when extracting coordinates of matching features from resampled images (e.g., in computer vision applications involving rectification). More research is needed.

**Response of complete pipeline** The reconstruction response plots in figures 10.2, though informative, show the response of only part of the pipeline. In contrast, figure 10.4 shows a visualization of the effect of the entire pipeline, for a variety of prefiltering and reconstruction strategies, including the effect of the sampling operation between them. In each plot, the blue curve shows what is left of the original $f$ in the reconstructed $\tilde{f}$. The purple curve includes the effect of all aliases generated by the sampling procedure. The shaded area between the curves therefore represents the amount of aliasing in the reconstructed signal. We focus on reconstruction with the box kernel, which is a reasonable approximation of typical LCD displays. One takeaway message is that a sharper prefilter preserves more of the original signal. Nonetheless, a softer prefilter may be preferable for reducing post-aliasing when a poor reconstruction filter shows excessive artifacts. Another interesting observation is that orthogonal projection with B-splines is very effective, except of course
Figure 10.3: Frequency response and phase errors in resampling. The shaded regions indicate the range of amplitude attenuation (or magnification) and phase errors introduced when resampling a signal under all possible translation offsets. These results corroborate those of figure 10.2. The cubic B-spline interpolation (b) performs better than other cubic kernels (d,e), and the quintic B-spline (c) performs significantly better than Lanczos (f).
Experiments and analyses

Figure 10.4: Frequency response for the complete pipeline. In each plot, the blue curve marks the combined effect of the prefilter and the reconstruction filter on a broad-spectrum function $f$ when no sampling is involved. The purple curve marks the effect of the entire pipeline, including sampling operations. The shaded region between them therefore represents aliasing. (a–g) reconstruct with the box filter, which is a good approximation for current LCD displays. (a) is orthogonal projection in space $V_{\beta^0}$. In (a–c), more content is preserved as prefilters become sharper, from the box to the cardinal cubic B-spline. (d) shows how Mitchell and Netravali [1988] softens results but also attenuates aliasing relative to (a–c). (e) shows the effect of directly sending the coefficients of orthogonal projection in $V_{\beta^3}$ for reconstruction with box, (f) shows the less objectionable idea of sampling the projection first, and (g) shows the more principled oblique projection from the cubic B-spline space to the box space. Finally, (h) shows the excellent performance of orthogonal projection in the cubic B-spline space. Note, however, the aliasing concentrated around the Nyquist rate.

when reconstructing with a different basis, in which case oblique projection is preferred. These experiments provide a glimpse of what is possible when orthogonal projection is realized in a space with good approximation properties. Example renderings for each of these pipelines are shown in the supplemental material.

Effect of repeated resampling The importance of good approximation properties in a sampling scheme is typically demonstrated using the cumulative effect of repeated resampling operations. Figure 10.5 shows the result of repeatedly translating or rotating an image 60 times. This challenging test may seem a somewhat artificial experiment. However, lower degradation rates are useful in real-time applications that amortize computations across subsequent frames (e.g., using reprojection). Furthermore, the test helps reveal filtering
Figure 10.5: Repeated resampling. A sharp image was rotated 60 times by 6° (top) and translated 60 times around a 5-pixel radius circle (bottom). We see that the amount of detail preserved is related to the approximation order. (a,b) The second-order hat and Mitchell-Netravali kernels perform worst, followed by the third-order Catmull-Rom (c). (d) Lanczos reconstruction ($L = 1$ with normalization, $W = 6$) causes certain frequencies to explode (see figures 10.2f and 10.3f). (e) The optimal-order cardinal cubic B-spline ($L = 4$) performs significantly better, but is surpassed by the remarkable cardinal cubic O-MOMS (f), which has the same approximation order but a smaller approximation constant.
Figure 10.6: Effect of repeated translations on PSNR. The top performing methods all include a digital filter $q$. We note the excellent behavior of interpolation with the cubic O-MOMS kernel ($W = 4$), and of approximation by the quadratic scheme of Condat et al. [2005] ($W = 3$). The Lanczos window is outperformed by the Hamming window ($W = 6$), and both are outperformed by the more efficient, high-approximation-order generalized kernels. Traditional linear interpolation, Mitchell and Netravali [1988], and Keys [1981] lag far behind.

Weaknesses that may not be visible in a single processing step but become apparent as part of a larger pipeline. Also, any blurring or contrast reduction is greatly accentuated and thus easier to visualize. It is remarkable how well the generalized methods perform relative to traditional approaches. The results also show the advantage of dropping the regularity constraint in favor of a lower approximation coefficient (see O-MOMS vs. B-spline).

Quantitative experiments In addition to providing results for visual inspection, we also present quantitative comparisons using both the traditional $L_2$ metric and the mean structural similarity (MSSIM) metric of Wang et al. [2004]. MSSIM is closer to the perceptual characteristics of the human visual system. Figure 10.6 shows a plot of the accumulated degradation, in terms of PSNR, as operations are repeated with a range of resampling strategies. Larger values of PSNR are ideal, and of course some loss in signal is inevitable due to the resampling operations. The MSSIM results are also summarized in table 10.1. Again, the generalized resampling strategies stand out for their improved reconstruction quality (for a given support size).
Table 10.1: Quantitative analysis of reconstruction quality. Kernel properties are degree $N$, width $W$, and approximation order $L$. The main columns report the mean structural similarity (MSSIM) between reference images (concentric circles as in figure 8.2 and average of four Kodak benchmarks) and their reconstructions for three types of experiments (repeated translations or rotations, and single upsampling). The kernels are sorted in descending order of average quality across all experiments. The interpolating B-splines $bspline^i$ consistently outperform the more traditional filters for the same $N$ and $W$. The O-MOMS kernels $omoms^*$ offer even slightly higher quality but at the expense of differentiability. The quasi-interpolant $condat^2$, which has degree 2 and support 3, also performs remarkably well.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>$N$</th>
<th>$W$</th>
<th>$L$</th>
<th>Properties</th>
<th>Translations</th>
<th>Rotations</th>
<th>Upscaling</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>omoms5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td>CIR</td>
<td>Kodak</td>
<td>CIR</td>
<td>Kodak</td>
</tr>
<tr>
<td>bspline5i</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
<td>0.993</td>
<td>0.965</td>
<td>0.999</td>
<td>0.983</td>
</tr>
<tr>
<td>omoms3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td>0.981</td>
<td>0.940</td>
<td>0.997</td>
<td>0.977</td>
</tr>
<tr>
<td>quasiiblu35</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td>0.965</td>
<td>0.919</td>
<td>0.994</td>
<td>0.973</td>
</tr>
<tr>
<td>condat3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td>0.962</td>
<td>0.915</td>
<td>0.991</td>
<td>0.969</td>
</tr>
<tr>
<td>hamming6</td>
<td>–</td>
<td>6</td>
<td>1</td>
<td></td>
<td>0.960</td>
<td>0.912</td>
<td>0.977</td>
<td>0.956</td>
</tr>
<tr>
<td>bspline3i</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td>0.948</td>
<td>0.900</td>
<td>0.977</td>
<td>0.957</td>
</tr>
<tr>
<td>condat2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td>0.930</td>
<td>0.884</td>
<td>0.989</td>
<td>0.966</td>
</tr>
<tr>
<td>lanczos6</td>
<td>–</td>
<td>6</td>
<td>1</td>
<td></td>
<td>0.964</td>
<td>0.828</td>
<td>0.987</td>
<td>0.960</td>
</tr>
<tr>
<td>bspline2i</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td>0.901</td>
<td>0.860</td>
<td>0.955</td>
<td>0.942</td>
</tr>
<tr>
<td>omoms2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td>0.814</td>
<td>0.802</td>
<td>0.961</td>
<td>0.945</td>
</tr>
<tr>
<td>sbaum2*</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td>0.822</td>
<td>0.804</td>
<td>0.921</td>
<td>0.921</td>
</tr>
<tr>
<td>lanczos4</td>
<td>–</td>
<td>4</td>
<td>1</td>
<td></td>
<td>0.822</td>
<td>0.804</td>
<td>0.896</td>
<td>0.911</td>
</tr>
<tr>
<td>keys†</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
<td>0.822</td>
<td>0.804</td>
<td>0.894</td>
<td>0.909</td>
</tr>
<tr>
<td>sbaum3‡</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td>0.822</td>
<td>0.804</td>
<td>0.876</td>
<td>0.902</td>
</tr>
<tr>
<td>dalai1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.657</td>
<td>0.716</td>
<td>0.956</td>
<td>0.941</td>
</tr>
<tr>
<td>linrev</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.686</td>
<td>0.716</td>
<td>0.960</td>
<td>0.928</td>
</tr>
<tr>
<td>condat1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.651</td>
<td>0.713</td>
<td>0.947</td>
<td>0.936</td>
</tr>
<tr>
<td>hamming4</td>
<td>–</td>
<td>4</td>
<td>1</td>
<td></td>
<td>0.663</td>
<td>0.718</td>
<td>0.822</td>
<td>0.878</td>
</tr>
<tr>
<td>mitchell1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td></td>
<td>0.581</td>
<td>0.680</td>
<td>0.625</td>
<td>0.806</td>
</tr>
<tr>
<td>linear</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>0.391</td>
<td>0.600</td>
<td>0.540</td>
<td>0.779</td>
</tr>
<tr>
<td>nearest</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0.042</td>
<td>0.279</td>
<td>0.547</td>
<td>0.717</td>
</tr>
</tbody>
</table>

*Same as IMOMS-2. †Same as Catmull-Rom. ‡Same as IMOMS-3. The kernels omoms* are from [Blu et al., 2001]; quasiiblu35 is from [Blu and Unser, 1999a]; bspline*i are the cardinal B-splines $\beta_{int}^i$; sbaum* are from [Schaum, 1993]; linrev is from [Blu et al., 2004]; condat* are from [Condat et al., 2005]; dalai1 is from [Dalai et al., 2006].
Discrete downscaling  Figure 10.7 shows a challenging example for discrete image downscaling using equation (5.28) (i.e., with no reconstruction). At the selected output resolution, the image is extremely prone to aliasing, which is clearly visible even with the soft Mitchell-Netravali kernel. In contrast, the cardinal cubic B-spline and O-MOMS kernels virtually eliminate aliasing, while maintaining sharpness indistinguishable from the wider Lanczos kernel.

Animation tests  In animation sequences, the effect of the resampling frequency response of section [10] is often visible. Phase errors cause high frequencies to oscillate in position, whereas amplitude errors cause high frequencies to oscillate in brightness. To demonstrate these effects, we use simple animation sequences in which each frame is the result of translating an input image around a circle with a 5 pixel radius, according to the naïve scheme of section [5.1]. These results are also available in the supplemental material.
Figure 10.7: Image downscaling using generalized sampling as in equation (5.28), using impulse reconstruction (i.e., \( r \ast \varphi = \delta \)), for various prefilters \( p \ast \psi \). The image shown in (a) was downscaled from its original (500 × 700) to an alias-prone resolution (65 × 91). It was then upscaled back to the original size using nearest neighbors to minimize any reproduction artifacts. (b,c) Note that nearest and box filtering result in a substantial amount of aliasing. (d,e) Mitchell-Netravali and Catmull-Rom (Keys) alleviate the problem, but aliasing is still clearly visible. (f) A wider support allows the Lanczos windowed-sinc to virtually eliminate aliasing while preserving image sharpness. (g,h) However, similar results are obtained with the cardinal cubic B-spline and O-MOMS filters, which have smaller support but include a fast digital filtering stage.
The framework of generalized sampling offers practical improvements in several areas of graphics. Our analysis and experiments confirm that it enables significantly higher quality in upsampling and resampling operations. We have adapted its concepts to derive a variety of supersampling techniques, and shown that these offer comparable quality to the best conventional techniques, but at a reduced computational cost. Similarly, we have explored a new family of image downscaling techniques, some of which outperform traditional algorithms in terms of blurring, ringing, and aliasing. Although generalized sampling does require solving discrete inverse filters, this computation is efficient and scales well to multicore and GPU architectures. Pedagogically, one of our contributions is a new parameterless notation that moves easily between discrete and continuous functions. By avoiding tedious index manipulations, this algebraic notations lets us to reason more intuitively about otherwise complex operations. We hope that these contributions may aid the dissemination of generalized sampling methods and attract broader interest from graphics researchers.
Acknowledgements

We would like to thank Leonardo Sacht for carefully reading the manuscript multiple times and providing insightful comments and corrections.
Appendices
The C++11 sample code in this appendix demonstrates the following set of functionalities:

- Fast inverse discrete convolution in lines 12–32.
- Safe indexing with reflection at boundaries in lines 34–38.
- Box kernel in lines 68–79.
- Hat kernel in lines 81–94.
- Catmull-Rom kernel in lines 138–149.
- Cardinal Cubic B-spline, a generalized kernel, in lines 159–171.
- Simple, intuitive upsampling in lines 202–220.
- Simple, intuitive downsampling in lines 222–251.
- Fast, incremental upsampling in lines 260–278.
- Fast, incremental downsampling in lines 280–307.
```cpp
#include <iostream> // std::ostream, std::cout
#include <cmath> // std::abs(), std::ceil(), std::floor()
#include <algorithm> // std::min()
#include <vector> // std::vector<>
#include <array> // std::array<>
#include <string> // std::string
#include <cassert> // assert()
#include <chrono> // std::chrono for timing benchmarks
using namespace std;

// Apply to sequence f the inverse discrete convolution given by
// a pre-factored LU decomposition

template<size_t M>
void linear_solve(const array<float,M>& L, vector<float>& f) {
    const int m = M, n = int(f.size());
    const float p_inv = 1.f; // Optimized for prescaled kernel where p = 1
    // Pre-factored decomposition only works when n>m. Grow sequence f if needed.
    while (f.size() <= m) {
        f.reserve(2*f.size()); // Prevent reallocation during insertions
        f.insert(f.end(), f.rbegin(), f.rend()); // Append a reflection
    }
    int nn = int(f.size()); // New size
    const float L_inf = L[m-1], v_inv = L_inf/(1.f+L_inf);
    // Forward pass: solve Lc' = f in-place
    for (int i=1; i<m; i++) f[i] -= L[i-1]*f[i-1];
    for (int i=m; i<nn; i++) f[i] -= L_inf *f[i-1];
    // Reverse pass: solve Uc' = c' in-place
    f[nn-1] *= p_inv*v_inv;
    for (int i=nn-2; i>=m-1; i--) f[i] = L_inf*(p_inv*f[i]-f[i+1]);
    for (int i=m-2; i>=0; i--) f[i] = L[i] *(p_inv*f[i]-f[i+1]);
    f.resize(n); // Truncate back to original size if grown
}

// Given sequence f, access f[i] using reflection at boundaries
static inline float get(const vector<float>& f, int i) {
    return i<0? get(f,-i-1): i>=int(f.size())?
        get(f,2*int(f.size())-i-1): f[i];
}

// Kernel interface
template<size_t N>
class KernelBase {
public:
    KernelBase() { b.fill(0.f); }
    // Evaluate kernel at coordinate x
    virtual float operator()(float x) const = 0;
};
```
virtual void digital_filter(vector<float>& f) const = 0;

int support() const { return N; }

virtual void accumulate_buffer(float fu, float u) = 0;

virtual float sample_buffer(float u) const = 0;

virtual string name() const = 0;

virtual float integral() const { return 1.f; }

protected:
array<float,N> b; // Incremental buffer

// Simple box kernel
struct Box final: KernelBase<1> {
float operator()(float x) const override {
    return x<=-0.5f || x>0.5f ? 0.f : 1.f;
}

void accumulate_buffer(float fu, float) override {
    b[0] += fu;
}

float sample_buffer(float) const override { return b[0]; }

void digital_filter(vector<float>&) const override { }

string name() const override { return "Box"; }
};

// Simple hat kernel
struct Hat final: KernelBase<2> {
float operator()(float x) const override {
    x = abs(x); return x>1.f ? 0.f : 1.f-x;
}

void accumulate_buffer(float fu, float u) override {
    b[0] += fu*(1.f-u); b[1] += fu*u;
}

float sample_buffer(float u) const override {
    return b[0]*(1.f-u)+b[1]*u;
}

void digital_filter(vector<float>&) const override { }
// Most cubics are $C^1$-continuous, symmetric, and have support 4.
// Factor out common functionality into a class.

template<typename Pieces>
class Symmetric4Pieces: public KernelBase<4> {
  public:
    float operator()(float x) const override final {
      x = abs(x); return x>2.f ? 0.f : x>1.f ? p.k0(2.f-x) : p.k1(1.f-x);
    }
    void accumulate_buffer(float fu, float u) override final {
      b[0] += fu*p.k3(u); b[1] += fu*p.k2(u);
      b[2] += fu*p.k1(u); b[3] += fu*p.k0(u);
    }
    float sample_buffer(float u) const override final {
      return b[0]*p.k3(u)+b[1]*p.k2(u)+b[2]*p.k1(u)+b[3]*p.k0(u);
    }
  private:
    Pieces p; // Polynomial pieces of kernel (k0:[-2,-1], k1:[-1,0], k2:[0,1], k3:[1,2])
};

// Traditional Mitchell-Netravali kernel
struct MitchellNetravaliPieces {
  static float k0(float u) {
    return (((7/18.f)*u-1/3.f)*u)*u;
  }
  static float k1(float u) {
    return (((-7/6.f)*u+1.5f)*u+.5f)*u+1/18.f;
  }
  static float k2(float u) {
    return (((7/6.f)*u-2.f)*u)*u+8/9.f;
  }
  static float k3(float u) {
    return (((-7/18.f)*u+5/6.f)*u-.5f)*u+1/18.f;
  }
};

struct MitchellNetravali final:
  Symmetric4Pieces<MitchellNetravaliPieces> {
    void digital_filter(vector<float>&) const override { }
    string name() const override { return "Mitchell-Netravali"; }
  };
struct CatmullRomPieces {
    static float k0(float u) { return ((.5f*u-.5f)*u)*u; }
    static float k1(float u) { return ((1.5f*u-2.5f)*u)*u+1.f; }
    static float k2(float u) { return ((.5f*u+1.f)*u-.5f)*u; }
};

struct CatmullRom final: Symmetric4Pieces<CatmullRomPieces> {
    void digital_filter(vector<float>&) const override { }
    string name() const override { return "Catmull-Rom"; }
};

// Cubic B-spline kernel pieces (multiplied by 6)
struct Bspline3Pieces {
    static float k0(float u) { return ((u)*u)*u; }
    static float k1(float u) { return ((-3.f*u+3.f)*u+3.f)*u+1.f; }
    static float k2(float u) { return ((3.f*u-6.f)*u)*u+4.f; }
    static float k3(float u) { return ((-u+3.f)*u-3.f)*u+1.f; }
};

// Generalized Cardinal Cubic B-spline kernel
struct CardinalBspline3 final: Symmetric4Pieces<Bspline3Pieces> {
    void digital_filter(vector<float>& f) const override {
        // Pre-factored L U decomposition of digital filter
        const array<float,8> L{.2f, .26315789f, .26760563f,
            .26792453f, .26794742f, .26794907f, .26794918f, .26794919f};
        linear_solve(L, f);
    }
    string name() const override {
        return "Cardinal Cubic B-spline";
    }
    float integral() const override { return 6.f; }
};

// Cubic OMOMS kernel pieces (multiplied by 5.25)
struct OMOMS3Pieces {
    static float k0(float u) {
        return ((.875f*u)*u+.125f)*u;
    }
    static float k1(float u) {
        return ((-2.625f*u+2.625f)*u+2.25f)*u+1.f;
    }
    static float k2(float u) {
        return ((2.625f*u-5.25f)*u+.375f)*u+3.25f;
    }
    static float k3(float u) {

return ((-.875f*u+2.625f)*u-2.75f)*u+1.f;
};

// Generalized Cardinal Cubic O-MOMS3 kernel
struct CardinalOMOMS3 final: Symmetric4Pieces<OMOMS3Pieces> {
    void digital_filter(vector<float>& f) const override {
        // Pre-factored L U decomposition of digital filter
        const array<float,9> L{.23529412f, .33170732f, .34266611f,
            .34395774f, .34411062f, .34412872f, .34413087f, .34413112f,
            .34413115f};
        linear_solve(L, f);
    }

    string name() const override { return "Cardinal Cubic O-MOMS"; }
    float integral() const override { return 5.25f; }
};

// Simple, intuitive implementation of upsampling
template<typename Kernel> static vector<float> upsample(vector<float> f, int m, Kernel& k) {
    assert(m >= int(f.size())); // Ensure we are upsampling
    vector<float> g(m); // New sequence of desired size m>f.size()
    k.digital_filter(f); // Apply kernel's associated digital filter
    const float kr = .5f*float(k.support());
    for (int j=0; j<m; j++) { // Index of sample in g
        float x = (j+.5f)/m; // Position in domain [0,1] of both f and g
        float xi = x*f.size()-0.5f; // Position in input sequence f
        int il = int(ceil(xi-kr)); // Leftmost sample under kernel support
        int ir = int(floor(xi+kr)); // Rightmost sample under kernel support
        double sum = 0.;
        for (int i=il; i<=ir; i++)
            sum += get(f,i)*k(xi-i);
        g[j] = float(sum);
    }
    return g;
}

// Simple, intuitive implementation of downsampling
template<typename Kernel> static vector<float> downsample(const vector<float>& f, int m, Kernel& k) {
    assert(m <= int(f.size())); // Ensure we are downsampling
    float s = float(m)/f.size(); // Scale factor
    const int n = int(f.size());
    const float kr = .5f*float(k.support());
    const bool should_normalize = (f.size()%m != 0);
    vector<float> g(m); // New sequence of desired size m<f.size()
for (int j=0; j<m; j++) {
    // Index of sample in g
    float x = (j+.5f)/m; // Position in domain [0,1] of both f and g
    int il = int(ceil ((x-kr/m)*n-.5f)); // Leftmost sample under kernel
    int ir = int(floor((x+kr/m)*n-.5f)); // Rightmost sample under kernel
    if (should_normalize) { // Should normalize?
        double sum = 0., sumw = 0.; // Sums of values and weights
        for (int i=il; i<=ir; i++) { // Loop over input samples
            float w = k((x-(i+.5f)/n)*m); // Weight for sample
            sum += w*get(f,i); sumw += w; // Accumulate values and weights
        }
        g[j] = k.integral()*float(sum/sumw); // Normalize by summed weights
    } else {
        for (int i=il; i<=ir; i++) {
            g[j] += k((x-(i+.5f)/n)*m)*get(f,i);
        }
        g[j] *= s;
    }
}
k.digital_filter(g); // Apply kernel's associated digital filter
return g;

// Advance to next sample
inline bool advance(int& i, int& j, double& u, double inv_s) {
    ++i; u += inv_s;
    if (u < 1.) return false;
    u -= 1.; ++j; return true;
}

// Faster, incremental implementation of upsampling
template<typename Kernel> vector<float> upsample2(vector<float> f, int m, Kernel& k) {
    k.digital_filter(f);
    double inv_s = double(f.size())/m; // Inverse scale factor
    // Output sample position between input samples
    double u = .5*(inv_s+(k.support()+1)%2);
    int fi = -k.support()/2-1;
    assert(f.size() <= m); // Ensure we are upsampling
    vector<float> g(m); // New sequence of desired size m>f.size()
    for (int i=0; i<k.support(); i++) // Initialize incremental buffer
        k.shift_buffer(get(f,++fi));
    for (int gi=0; gi<m; ) { // Sample reconstruction of f into g[gi]
        g[gi] = k.sample_buffer(u);
        if (advance(gi, fi, u, inv_s))
            k.shift_buffer(get(f,fi));
        }
return g;
}

// Faster, incremental implementation of downsampling
template<typename Kernel> vector<float> downsample2(const vector<float>& f, int m, Kernel& k) {
    const int n = f.size();
    double s = double(m)/n; // Scale factor
    double kr = .5*k.support();
    int fi = int(ceil(((.5-kr)/m)*n-.5));
    // Input sample position between output samples
    double u = ((fi+.5)/n)*m -(.5-kr);
    assert(f.size() >= m); // Ensure we are downsampling
    vector<float> g(m); // New sequence of desired size m>f.size()
    int gi = -k.support();
    for (int i=0; i < k.support(); i++) // Initialize incremental buffer
        k.shift_buffer(0.f);
    while (gi < -1) {
        k.accumulate_buffer(get(f, fi), u);
        if (advance(fi, gi, u, s)) k.shift_buffer(0.f);
    }
    while (1) {
        k.accumulate_buffer(get(f, fi), u);
        if (advance(fi, gi, u, s)) {
            if (gi >= m) break;
            g[gi] = s*k.shift_buffer(0.f);
        }
    }
    k.digital_filter(g);
    return g;
}

// Output a sequence
static ostream& operator<<(ostream& out, const vector<float>& f) {
    for (int i=0; i<f.size(); i++)
        out << (i+.5)/f.size() << ' ' << f[i] << '
';
    return out;
}

// Report elapsed lifetime of object
class Timing {
public:
    Timing(const string& name) : m_name(name), m_start(now()) { }
    ~Timing() {
        using namespace std::chrono;
        double t = duration_cast<microseconds>(now()-m_start).count();
        cout << m_name << "	" << t << " ms" << endl;
    }
};

Source-code

78
cout << m_name << " in " << t/1000. << " ms\n";
}

private:
    using clock = chrono::high_resolution_clock;
    using time_point = chrono::time_point<clock>;
    static time_point now() { return clock::now(); }
    string m_name;
    time_point m_start;
};

static double
maxerr(const vector<float>& f, const vector<float>& g) {
    double m = 0.;
    assert(f.size() == g.size());
    for (size_t i=0; i<f.size(); i++) {
        m = max(m, abs(double(g[i]) - f[i]));
    }
    return m;
}

template<typename Kernel>
void test_performance(int up, int down) {
    Kernel k;
    vector<float> f{0.f, 3.f, 1.f, .5f, 4.f, 2.f}, g(f);
    { Timing t(k.name() + " up"); f = upsample(f, up, k); }
    { Timing t(k.name() + " up2"); g = upsample2(g, up, k); }
    cout << " " << maxerr(f, g) << " max error\n";
    { Timing t(k.name() + " down"); f = downsample(f, down, k); } // shows downsample in debug
    { Timing t(k.name() + " down2"); g = downsample2(g, down, k); }
    cout << " " << maxerr(f, g) << " max error\n";
}

// Plot kernel impulse response. Pipe through gnuplot -persist
template <typename Kernel>
static void
plot(const vector<float>& f, int window, Kernel& k, int w) {
    cout << "set terminal aqua " << window << "; // Change to your needs"
    cout << "set title " << k.name() << "\n";
    cout << "set xrange [" << -.5f*w << ";" << .5f*w << "]\n";
    cout << "plot \"-\" u (" << w << "*($1-0.5)):2 w 1 t \"\"
";
    cout << f << "e\n";
}

template <typename Kernel>
void plot(int up, int down, int& window) {
    Kernel k;
}
// Trick to see impulse response
vector<float> f{0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f};
const int w = f.size();
f = upsample(f, up, k);
gnuplot(f, window++, k, w);
f = downsample(f, down, k);
gnuplot(f, window++, k, w);
}

template<typename Kernel>
static void check_interpolation() {
    Kernel k;
cout << "checking " << k.name() << 'n';
vector<float> f{.5f, 2.f, 1.f, 0.f, 5.f};
vector<float> g = upsample(f, f.size(), k);
cout << " " << maxerr(f, g) << " max error\n";
g = downsample(f, f.size(), k);
cout << " " << maxerr(f, g) << " max error\n";
}

int main() {
    if (1) {
        const int up = 1000001, down = 101;
test_performance<Box>(up, down);
test_performance<Hat>(up, down);
test_performance<CatmullRom>(up, down);
test_performance<MitchellNetravali>(up, down);
test_performance<CardinalBspline3>(up, down);
test_performance<CardinalOMOMS3>(up, down);
    } else if (0) {
        const int up = 1001, down = 51;
        int window = 1;
    plot<Box>(up, down, window);
    plot<Hat>(up, down, window);
    plot<CatmullRom>(up, down, window);
    plot<MitchellNetravali>(up, down, window);
    plot<CardinalBspline3>(up, down, window);
    plot<CardinalOMOMS3>(up, down, window);
    } else {
        check_interpolation<Box>();
        check_interpolation<Hat>();
        check_interpolation<CatmullRom>();
        check_interpolation<CardinalBspline3>();
        check_interpolation<CardinalOMOMS3>();
    }
}


Bibliography


Bibliography


