Fast High-Dimensional Filtering Using the Permutohedral Lattice

Andrew Adams, Jongmin Baek, Myers Abraham Davis

Stanford University

Abstract

Many useful algorithms for processing images and geometry fall under the general framework of high-dimensional Gaussian filtering. This family of algorithms includes bilateral filtering and non-local means. We propose a new way to perform such filters using the permutohedral lattice, which tessellates high-dimensional space with uniform simplices. Our algorithm is the first implementation of a high-dimensional Gaussian filter that is both linear in input size and polynomial in dimensionality. Furthermore it is parameter-free, apart from the filter size, and achieves a consistently high accuracy relative to ground truth (> 45 dB). We use this to demonstrate a number of interactive-rate applications of filters in as high as eight dimensions.

Categories and Subject Descriptors (according to ACM CCS): I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering

1. Introduction

High-dimensional Gaussian filtering (Equation 1) is a powerful way to express a smoothness prior on data in an arbitrary Euclidean space. As such, it is an important component of many algorithms in image processing and computer vision. Algorithms such as color or grayscale bilateral filtering [AW95] [SB97] [TM98], joint bilateral filtering [ED04] [PSA*04], joint bilateral upsampling [KCLU07], non-local means [BCM05], and spatio-temporal bilateral filtering [BM05] can all be expressed as high-dimensional Gaussian filters.

Recent work on accelerating high-dimensional Gaussian filters has focused on explicitly representing the high-dimensional space with point samples, using a regular grid of samples [PD09] or a cloud of samples stored in a kd-tree [AGDL09]. When the space is explicitly represented in this way, filtering is implemented by resampling the input data onto the high-dimensional samples, performing a high-dimensional Gaussian blur on the samples, and then resampling back into the input space (Figure 1). [AGDL09] terms these three stages splatting, blurring, and slicing.

We propose accelerating such filters by sampling the high-dimensional space at the vertices of the permutohedral lattice (illustrated in Figure 2). The lattice is composed of identical

$$\vec{v}_i{}' = \sum_{j=1}^{n} e^{-\frac{1}{2}|\vec{p}_i-\vec{p}_j|^2}\vec{v}_j$$

Equation 1: *High-dimensional Gaussian filtering associates an arbitrary position $\vec{p}_i$ with each value $\vec{v}_i$ to be filtered, and then mixes values with other values that have nearby positions. Usually the values are homogeneous pixel colors. If the positions are two-dimensional pixel locations, then this expresses a Gaussian blur. If the positions are pixel locations combined with color, for a total of five dimensions, this expresses a color bilateral filter. If the position vectors are derived from local neighborhoods around each pixel, then this expresses non-local means.*

simplices (high-dimensional tetrahedra), and the enclosing simplex of any given point can be found by a simple rounding algorithm. Splatting and slicing can therefore be done by barycentric interpolation, which is exponentially cheaper than the multi-linear interpolation of [PD09], and does not suffer from the irregularity and parameter-heavy nature of the randomly bifurcating kd-tree queries of [AGDL09]. Similarly to the grid approach, the blurring stage can be done with a simple separable filter. We describe the lattice and its properties in section 3.

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

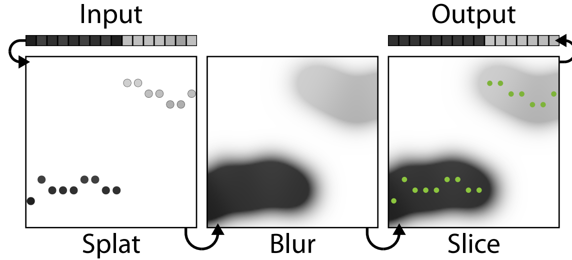

Figure 1: *High-dimensional Gaussian filtering can be implemented by first embedding the input values $\vec{v}$ at positions $\vec{p}$ in a high-dimensional space (splatting), then performing a Gaussian blur in that space (blurring), then sampling the space at the original positions $\vec{p}$ (slicing). The diagram above illustrates a bilateral filter of a one-dimensional signal using this framework.*

Using the permutohedral lattice for high-dimensional filtering of n values in d dimensions has a time complexity of $O(d^2n)$ and a space complexity of $O(dn)$. This compares favorably to existing techniques, and in fact this method proves to be faster than the state of the art for a wide range of filter sizes and dimensionalities (Figure 7). We describe its performance and accuracy in section 4. The run-time is sufficiently low that we can demonstrate the first real-time color bilateral filter (a five-dimensional filter). The qualitative change brought about by this speed increase is that we enable interactive use of this class of filter, described in section 5.

2. Prior Work

High-dimensional Gaussian filtering includes, as special cases, the bilateral filter, and the non-local means filter. In turn, high-dimensional Gaussian filtering is a type of Gauss transform, in which a weighted sum of Gaussians, centered at points $\vec{p}_i$, is sampled at some other locations $\vec{q}_i$. For our case (Equation 1), the weights are always homogeneous vectors, usually storing color, and $\vec{p}_i$ typically equals $\vec{q}_i$.

There are thus several categories of related work: Bilateral filters and attempts to accelerate them; non-local means and corresponding accelerations; and more general attempts to accelerate computation of the fast Gauss transform. We will discuss each in turn.

2.1. The Bilateral Filter

Bilateral filtering [AW95] [SB97] [TM98] averages pixels with other pixels that are nearby in both position and intensity. It is a five-dimensional Gaussian filter (Eq. 1) in which $\vec{v}_i = [r_i, g_i, b_i, 1]$ (the homogeneous color at pixel i), and $\vec{p}_i = [\frac{x_i}{\sigma_s}, \frac{y_i}{\sigma_s}, \frac{r_i}{\sigma_c}, \frac{g_i}{\sigma_c}, \frac{b_i}{\sigma_c}]$, where σ_s is the spatial standard deviation of the filter and σ_c is the color-space standard deviation. Grayscale bilateral filtering can be expressed by replacing r_i, g_i, and b_i with just luminance, and is hence a three-dimensional filter. Some notable uses of the bilateral filter include tone mapping [DD02], halo-free sharpening, and photographic tone transfer [BPD06].

Bilateral filters of dimensionalities other than 3 and 5 also occur. Weber et al. [WMM*04] use a four-dimensional bilateral filter with a temporal term for smoothing photon density maps for rendering ray-traced sequences. Adams et al. [AGDL09] similarly add a temporal term to a color bilateral filter to denoise video, which results in a 6-D filter.

Joint Bilateral Filters By deriving $\vec{p}_i$ from one image and $\vec{v}_i$ from another, one can smooth an image in a way that does not cross the edges of another. This technique was invented independently by Eisemann and Durand [ED04] and Petschnigg et al. [PSA*04] and was used by each for combining images taken with and without flash.

This idea was extended by Kopf et al. [KCLU07] to use as an upsampling technique. By splatting at positions determined by the low-resolution input, and then slicing using a high-resolution reference image, the low resolution data can be interpolated in a way that respects edges in the high-resolution reference.

Fast Bilateral Filters Given the wide ranging utility of the bilateral filter, considerable research has gone into accelerating it. Durand and Dorsey [DD02] discretize in intensity, and compute a regular Gaussian blur at each intensity level using an FFT, which can then be interpolated between to produce a grayscale bilateral filter. This approach evolved into the bilateral grid [PD06] of Paris and Durand, which was implemented on a GPU for various interactive applications by Chen et al. [CPD07]. The bilateral grid discretizes in space and intensity, and introduced the splat-blur-slice pipeline for high-dimensional filtering (Figure 1). Paris and Durand also describe a five-dimensional grid for color bilateral filtering [PD09].

The work of Durand and Dorsey [DD02] was independently extended by Porikli [Por08] who observed that the FFTs were unnecessary, and used faster integral-image based methods to blur each intensity level, thus producing integral histograms. Yang et al. [YTA09] improve on this by not explicitly representing the entire space, but instead sweeping a plane through the intensity levels, computing the output in intensity order. Their low-memory, cache-friendly algorithm is the fastest known grayscale bilateral filter. However, the plane-sweep approach does not generalize well to higher dimensions.

2.2. Non-Local Means

Bennett and McMillan [BM05], while denoising video with a combination of bilateral filters, noted that the positions $\vec{p}_i$ could be usefully augmented by including the color or intensity of nearby pixels as well. This is equivalent to non-local

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

means of Buades et al. [BCM05] [BCM08], which is an optimal denoiser for additive white noise.

Fast Non-Local Means There are numerous methods for accelerating non-local means that do not explicitly represent the high-dimensional space. For example, Mahmoudi and Sapiro [MS05] preclassify regions of the image according to average intensity and gradient direction in order to restrict the search, whereas Darbon et al. [DCC*08] compute integral images of certain error terms, as do Wang et al. [WGY*06]. However, this family of methods is restricted to position vectors composed of rectangular image patches, and does not generalize to less structured denoising tasks, such as the geometry denoising [AGDL09] and the aforementioned photon density denoising [WMM*04].

Fast Gauss Transforms Approaches for evaluating Equation 1 as a generic Gauss transform have focused on tree-based representations of the high-dimensional space of position vectors. Adams et al. use a kd-tree coupled with randomized queries, whereas Brox et al. [BKC08] use a cluster tree.

Yang et al. [YDGD03] describe the improved fast Gauss transform, which uses a cluster tree in which each leaf stores not only a value, but also some Taylor series expansion terms describing how the value varies about that leaf. It is extremely accurate, but not as fast as other methods.

3. The Permutohedral Lattice

We now describe the lattice we will use to accelerate high-dimensional Gaussian filters. The d-dimensional permutohedral lattice is the projection of the scaled regular grid $(d+1)\mathbb{Z}^{d+1}$ along the vector $\vec{1} = [1,\ldots,1]$ onto the hyperplane $H_d : \vec{x}.\vec{1} = 0$, which is the subspace of $\mathbb{R}^{d+1}$ in which coordinates sum to zero. It is hence spanned by the projection of the standard basis for $(d+1)\mathbb{Z}^{d+1}$ onto H_d:

$$B_d = \begin{pmatrix} d & -1 & \ldots & -1 \\ -1 & d & \ldots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \ldots & d \end{pmatrix}$$

Each of the $(d+1)$ basis vectors (columns of B_d) has coordinates that sum to zero, and that each coordinate of each basis vector has a consistent remainder modulo $d+1$. Both of these properties are preserved when taking integer combinations, so points in the lattice are those points with integer coordinates that sum to zero and have a consistent remainder modulo $d+1$. We describe a lattice point whose coordinates have a remainder of k as a "remainder-k" point. In Figure 2 we show the lattice for $d=2$, and label each lattice point by its remainder.

The permutohedral lattice has several properties that make

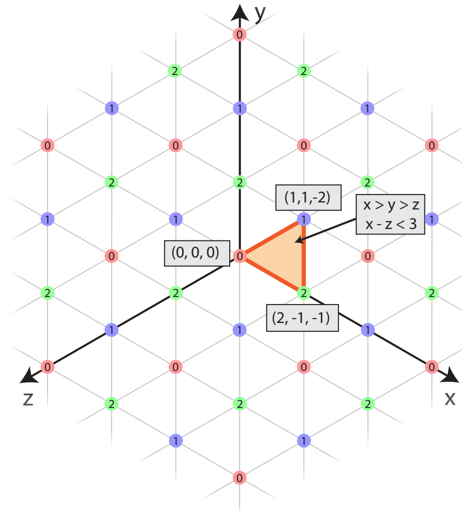

Figure 2: *The d-dimensional permutohedral lattice is formed by projecting the scaled grid $(d+1)\mathbb{Z}^{d+1}$ onto the plane $\vec{x}\cdot\vec{1} = 0$. This forms the lattice $(d+1)A_d^*$, which we term the permutohedral lattice, as it describes how to tile space with permutohedra. Lattice points have integer coordinates with a consistent remainder modulo $d+1$. In the diagram above, which illustrates the case $d=2$, points are labeled and colored according to their remainder. The lattice tessellates the plane with uniform simplices, each simplex having one vertex of each remainder. The simplices are all translations and permutations of the canonical simplex (highlighted), which is defined by the inequalities $x_0 > x_1 > \ldots > x_d$ and $x_0 - x_d < d+1$.*

it well-suited for high-dimensional filtering using the splat-blur-slice pipeline illustrated in Figure 1. Firstly, it tessellates high-dimensional space with uniform simplices, so we can use barycentric interpolation to splat the signal onto the lattice points. Secondly, the enclosing simplex of any point, along with its barycentric coordinates, can be computed quickly (in $O(d^2)$ time), so splatting is fast. Thirdly, the neighbors of a lattice point are trivial to compute, so the blur stage is also fast. Finally, slicing can be done using the barycentric weights already computed during the splatting stage, and so is also fast. These properties are described briefly below. Full derivations, proofs, and further helpful properties can be found in [BA09].

The permutohedral lattice tessellates H_d with uniform simplices. Consider the d-dimensional simplex whose vertices $\vec{s}_0,\ldots,\vec{s}_d$ are given by:

$$\vec{s}_k = [\underbrace{k,\ldots,k}_{d+1-k},\ \underbrace{k-(d+1),\ldots,k-(d+1)}_{k}]$$

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

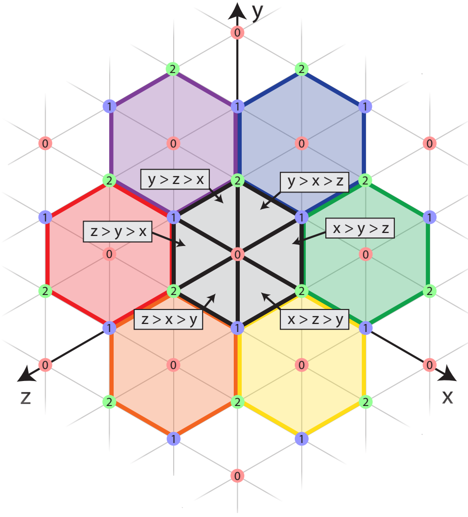

Figure 3: *When using the permutohedral lattice to tessellate the subspace H_d, any point $\vec{x} \in H_d$ is enclosed by a simplex uniquely identified by the nearest remainder-0 lattice point $\vec{l}_0$ (the zeroes highlighted in red) and the ordering of the coordinates of $\vec{x} - \vec{l}_0$. The nearest remainder-0 lattice point can be computed with a simple rounding algorithm, and so identifying the enclosing simplex of any point and enumerating its vertices is computationally cheap ($O(d^2)$).*

We term this simplex the canonical simplex. For example, when $d = 4$ the vertices are the columns of:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & -1 \\ 0 & 1 & 2 & -2 & -1 \\ 0 & 1 & -3 & -2 & -1 \\ 0 & -4 & -3 & -2 & -1 \end{pmatrix}$$

Note that $\vec{s}_k$ is a lattice point of remainder k (i.e. its coordinates are all congruent to k modulo $d+1$), and the simplex includes one point of each remainder. The vertices of this simplex are the boundary cases of the inequalities $x_0 \geq x_1 \geq \ldots \geq x_d$ and $x_0 - x_d \leq d+1$, and a point lies within the simplex if and only if it obeys these inequalities.

Now consider any permutation ρ of the coordinates of the canonical simplex. Each ρ induces a corresponding ordering of the coordinates $x_{\rho(0)} \geq x_{\rho(1)} \geq \ldots \geq x_{\rho(d)}$, and the inequality $x_{\rho(0)} - x_{\rho(d)} < d+1$. Taking the union of these inequalities across all $(d+1)!$ simplices results in the set $\{x_i | \max_i x_i - \min_i x_i \leq d+1\}$ (the central hexagon in Figure 3), which is in fact the set of all points which have the origin as their closest remainder-0 point (See Proposition A.1).

Hence, as the lattice is translation invariant, a point $\vec{x} \in H_d$, with closest remainder-0 point $\vec{l}_0$, belongs to a unique simplex determined by $\vec{l}_0$ and the ordering of the coordinates of $\vec{l}_0 - \vec{x}$ (Figure 3). As every point belongs to a unique simplex, which is a permutation and translation of the canonical simplex, H_d is tessellated by uniform simplices.

The vertices of the simplex containing any point in H_d can be computed in $O(d^2)$ time. This property will be useful for the splat and slice stages of filtering.

The vertices of the simplex containing some point $\vec{x} \in H_d$ can be generated by first computing the closest remainder-0 point $\vec{l}_0$, and then sorting the difference $\vec{l}_0 - \vec{x}$. The resulting permutation and translation can then be applied to the canonical simplex to compute the simplex vertices in $O(d^2)$ operations.

The closest remainder-0 point can be found by first rounding each coordinate of $\vec{x}$ to the nearest multiple of $(d+1)$, and then, if the result is outside the subspace H_d, greedily walking back to H_d by rounding those coordinates that moved the farthest in the other direction instead. The sublattice formed by the remainder-0 points is called A_{d+1}, and this is the algorithm given by Conway and Sloane ([CS99] pp 446) for finding the closest point in that lattice.

The nearest neighbors of a lattice point can be computed in $O(d^2)$ time. This property will be useful during the blur stage of filtering.

The basis vectors given by B_d above are those of minimal length, so the nearest neighbors of a lattice point $\vec{l}_k$ are those separated by a vector of the form $\pm[-1, \ldots, -1, d, -1, \ldots, -1]$. The are $2(d+1)$ such neighbors, and each is described by a vector of length $d+1$, and so the neighbors can be fully enumerated in $O(d^2)$ time.

3.1. Filtering using the Lattice

There are four main stages for using the permutohedral lattice for high-dimensional Gaussian filtering, illustrated in Figure 4. We will describe each in turn.

Generating position vectors. Firstly, the position vectors $\vec{p}_i$, representing the locations in the high-dimensional space, must be generated and embedded in H_d. Generating the positions is somewhat application dependent. For a color bilateral filter, we generate 5-D position vectors of the form $[\frac{x_i}{\sigma_p}, \frac{y_i}{\sigma_p}, \frac{r_i}{\sigma_c}, \frac{g_i}{\sigma_c}, \frac{b_i}{\sigma_c}]$, by augmenting the input image with two extra channels encoding spatial location, and then scaling each channel by the inverse of the desired standard deviation. For non-local means we would instead either extract local windows around each pixel, or compute some bank of filters around each pixel and record the responses. Typically we do the latter, using PCA to determine the optimal filter bank, as first proposed by Tasdizen [Tas08].

We must then scale the position vectors by the inverse of the standard deviation of the blur induced by the remaining steps, which totals $\sqrt{\frac{2}{3}}(d+1)$ in each dimension (derived

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

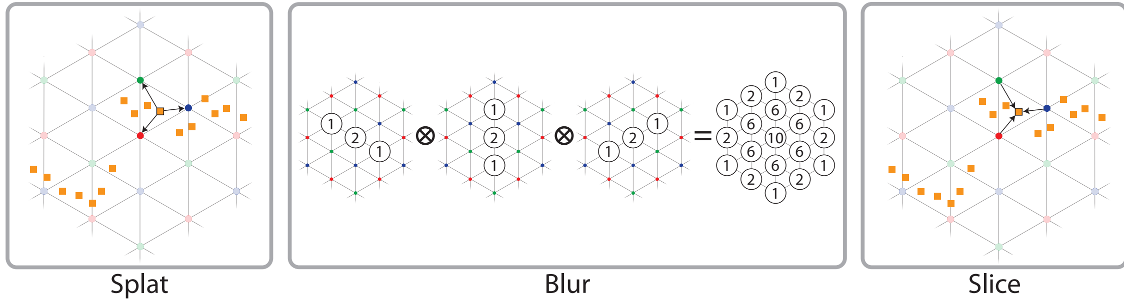

Figure 4: *To perform a high-dimensional Gaussian filter using the permutohedral lattice, first the position vectors $\vec{p}_i \in \mathbb{R}^d$ are embedded in the hyperplane H_d using an orthogonal basis for H_d (not pictured). Then, each input value **splats** onto the vertices of its enclosing simplex using barycentric weights. Next, lattice points **blur** their values with nearby lattice points using a separable filter. Finally, the space is **sliced** at each input position using the same barycentric weights to interpolate output values.*

below). Next we embed the position vectors in the subspace H_d. The basis for H_d given above is unsuitable for this task because it is not orthogonal, so we instead use the orthogonal basis:

$$E = \begin{pmatrix} 1 & 1 & \dots & 1 \\ -1 & 1 & \dots & 1 \\ 0 & -2 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -d \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{6}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sqrt{d(d+1)}} \end{pmatrix}$$

We choose this basis because it allows us to compute $E\vec{x}$ in O(d) time using the recurrence:

$$\begin{aligned} (E\vec{x})_d &= -\alpha_d x_{d-1} \\ (E\vec{x})_i &= -\alpha_i x_{i-1} + x_i/\alpha_{i+1} + (E\vec{x})_{i+1} \\ (E\vec{x})_0 &= x_i/\alpha_1 + (E\vec{x})_1 \\ \alpha_i &= \sqrt{i/(i+1)} \end{aligned}$$

Splatting. Once each position has been embedded in the hyperplane, we must identify its enclosing simplex and compute barycentric weights. The enclosing simplex of any point can be described by the permutation and translation that maps the simplex back to the canonical simplex, which can be computed in $O(d \log d)$ by using the rounding algorithm described earlier to find the nearest remainder-0 point, and then sorting the residual.

Therefore, to compute barycentric coordinates for a point $E\vec{p}_i$ in an arbitrary simplex, we can apply the translation and permutation to map $E\vec{p}_i$ to some $\vec{y}$ within the canonical simplex. Barycentric coordinates $\vec{b}$ for $\vec{y}$ are then given by the expression below, as shown by Proposition A.2.

$$b_0 = 1 - \sum_{j=1}^{d} b_j, \qquad b_i = \frac{y_{d-i} - y_{d-i+1}}{d+1}$$

Barycentric interpolation is invariant to translation and permutation, and so these barycentric coordinates for $\vec{y}$ within the canonical simplex are also the barycentric coordinates for $E\vec{p}_i$ within its simplex. Once the barycentric weights are computed, $b_k\vec{v}_i$ is added to the value stored at the remainder-k lattice point in the enclosing simplex of $\vec{p}_i$ (recall that $\vec{v}_i$ is the homogeneous value associated with position $\vec{p}_i$). The lattice point values are stored in a hash table. Lattice points that do not yet exist in the hash table are created when they are first referred to during the splat stage, and start with an initial value of zero.

There are two ways to identify each lattice point for use as a hash table key. One can apply the inverse permutation and translation to the remainder-k point of the canonical simplex to compute the lattice point's position, and use that as a key. Each key is a vector of length $d+1$, and so this results in a memory complexity of $O(dl)$ for l lattice points.

In rare cases where $l > n$, we can alternatively achieve a memory complexity of $O(dn)$ for n input values by separately storing the simplex enclosing each input position $\vec{p}_i$, as a simplex can be identified uniquely in $O(d)$ memory by its remainder-0 point and its permutation. We then identify a lattice point using its remainder and a pointer to any simplex it belongs to, for an additional $O(dn)$ memory. One lattice point belongs to many simplices, so key comparison is done by using the simplex and remainder to compute the lattice point's coordinates on the fly.

l is loosely bounded by $O(dn)$, as each input value creates at most $d+1$ new lattice points. However, filters near this bound correspond to very small filter sizes and are not very useful, as no shared lattice points means no cross-talk between pixels, and hence no filtering. In practice, we find that $l < n$, so we prefer the first, faster scheme. In either case,

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

each hash table access costs $O(d)$ time for key comparison. Each pixel accesses the hash table $O(d)$ times, and so the time complexity of splatting is $O(d^2 n)$.

Barycentric interpolation in the permutohedral lattice is equivalent to convolution by the projection of a uniformly-weighted $(d+1)$-dimensional hypercube of side length $d+1$ onto H_d, and induces a total variance of $d(d+1)^2/12$ (See Proposition A.3.)

Blurring. Now that our values are embedded in the subspace H_d, the next stage of high-dimensional Gaussian filtering is performing a regular Gaussian blur within that subspace. To do this we convolve by the kernel $[1\ 2\ 1]$ along each lattice direction of the form $\pm[1,\ldots,1,-d,1,\ldots,1]$ (Figure 4). This produces an approximate Gaussian kernel with total variance $d(d+1)^2/2$.

The blur stage spreads energy from each lattice point to $O(3^d)$ neighbors. If we created hash table entries for new lattice points reached during the blur then the memory use would grow quite large. We therefore do not create new lattice points during the blur phase, which incurs some accuracy penalty relative to a naive computation of Equation 1, as points that may have transferred energy could instead belong to disconnected regions of the lattice.

Adams et al. [AGDL09] observe a similar effect, and argue it may actually be advantageous. For example, when bilateral filtering, the absence of these "stepping-stone" lattice points will prevent energy transfer from a white pixel to a black pixel across a hard edge, but will allow energy transfer between a black pixel and a white pixel on either side of a smooth gradient.

The blur step involves looking up $O(d)$ neighbors for each lattice point. Each lookup takes $O(d)$ time for hash table key comparison, and so the blur step has time complexity $O(d^2 l)$.

Slicing. Slicing is identical to splatting, except that it uses the barycentric weights to gather from the lattice points instead of scattering to them. It produces the same total variance of $d(d+1)^2/12$, which brings the total variance induced by the algorithm to $\frac{2}{3}d(d+1)^2$, which is equivalent to a standard deviation in each dimension of $\sqrt{\frac{2}{3}}(d+1)$. Slicing can be accelerated by storing the barycentric weights and pointers to lattice point values computed during splatting. This "slicing table" can then be scanned through in $O(dn)$ time to slice. The entire algorithm thus has a time complexity of $O(d^2(n+l))$.

3.2. GPU Implementation

The above algorithm is fairly straightforward to parallelize on a GPU. We constructed an implementation using NVIDIA's CUDA [Buc07], and achieve typical speedups of 6x on a Geforce GTX280 compared to the single-threaded CPU implementation on an Intel Core i7 920. Clearly a hash-table benefits from a cache.

Figure 5: *At the top is a 512x256 crop of the input image used for time and memory comparisons - a typical 1.5 megapixel photograph. Below is the same crop of the output of the permutohedral lattice used to perform a color bilateral filter with a spatial standard deviation of* 16 *pixels and a color standard deviation of* $\frac{1}{8}$. *It is visually indistinguishable from the naive result. The permutohedral lattice produces a result with a PSNR between 45 and 50 dB relative to an exhaustive evaluation of Equation 1, depending on filter size and dimensionality.*

The main point of difference between the CPU and GPU versions is related to the creation of hash table entries during the splatting stage. It is customary to attach locks to hash table entries and synchronize all accesses to a given entry to prevent erroneously inserting one key in multiple places. We found it faster to break the splatting stage into three. First, we compute the slicing table, recording which lattice points each input pixel splats to, and with what weights. While doing this, we insert the lattice points found into the hash table in a way which permits individual keys being inserted in multiple locations. Specifically, while we still lock each hash table entry before insertion, other simultaneous hash table insertions simply skip over locked entries while looking for a free spot rather than waiting on the lock to see if the key matches. This means we never have data dependencies involving one query reading the key that another query has written, so we can write the keys using faster non-atomic writes, and only the smaller array of locks needs to be coherent. Next, we rehash the entries of the slicing table and update it so that every reference to a lattice point refers to the unique earliest instance of that lattice point in the hash

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

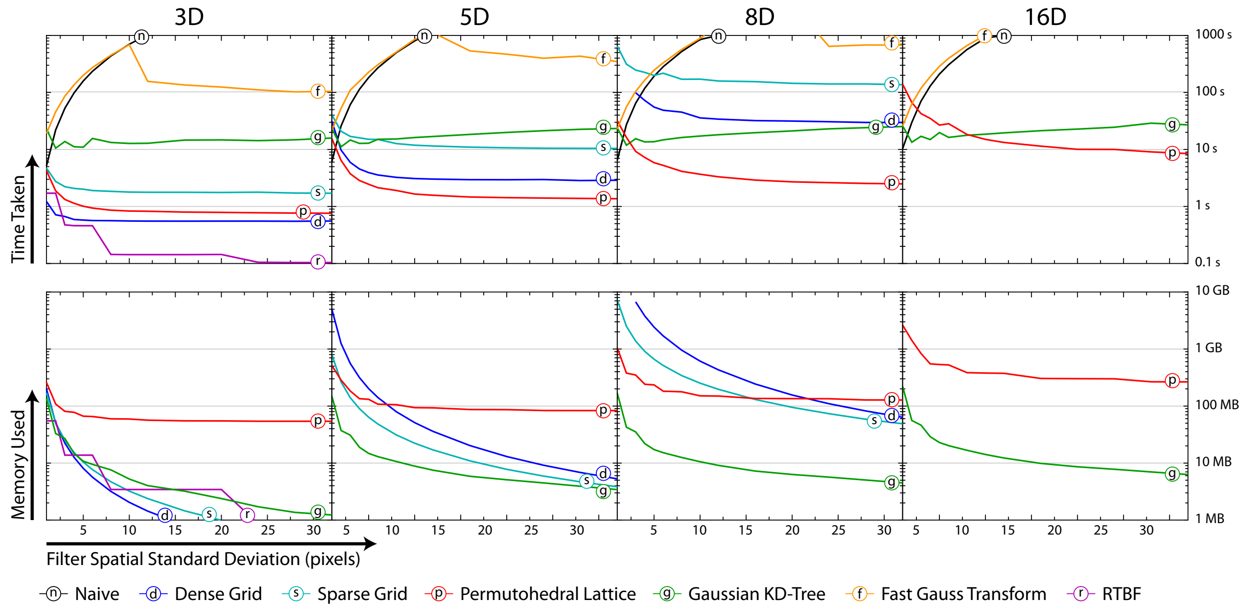

Figure 6: *Here we show time taken and memory used as a function of filter size and dimensionality for a variety of algorithms. The input was a typical 1.5 megapixel photograph (Figure 5). Position vectors were composed of pixel locations, pixel luminance, pixel chrominance, and then increasing numbers of filter responses over a Gaussian-weighted 9x9 local window, depending on the desired dimensionality. These filters were derived using PCA on the space of such local windows, and are typically local derivatives. The standard deviation for all but the spatial terms is fixed at $\frac{1}{8}$. The methods compared are the naive windowed bilateral filter (n), the dense bilateral grid (d) [CPD07], the sparse bilateral grid (s), the permutohedral lattice (p), the Gaussian KD-Tree (g) [AGDL09], the improved fast Gauss transform (f) [YDGD03], for which memory use is unavailable, and real-time bilateral filtering (r) [YTA09], which does not apply to dimensionalities above three. The parameters of each algorithm were tuned so that it ran as quickly as possible while achieving a PSNR of roughly 45dB relative to an exhaustive evaluation of Equation 1.*

table. Finally, we use the corrected slicing table to splat, additively scattering onto lattice points as usual.

In an interactive setting, it is common to perform many filters whilst only changing a few parameters in between runs. If the position vectors and filter sizes do not change, then the old slicing table and hash table can be reused for a moderate speedup.

4. Results and Comparisons

Here we present results of a comparison of a number of algorithms for performing high-dimensional filtering on a typical photograph. All algorithms were run single-threaded on an Intel Core i7 920 running at 2.67 GHz. Only algorithms with source code available were used, and all low-level optimizations were left up to the compiler. The input photograph and the output of the permutohedral lattice are shown in Figure 5. The running times and memory use of each algorithm for a few different dimensionalities are shown in Figure 6. Figure 7 illustrates the fastest method for a variety of dimensionalities and filter sizes. The permutohedral lattice is significantly faster than the state of the art for a large range of dimensionalities and filter sizes.

4.1. The Algorithms

Naive: This algorithm considers all pixels within three spatial standard deviations of each pixel and manually computes Equation 1. While this algorithm scales linearly with dimensionality, it is quadratic in filter size, and does not work for unstructured data such as geometry.

Dense Grid: This is the bilateral grid of Paris et al. [PD09], using multi-linear splatting and slicing, and a separable blur kernel of $[1\ 2\ 1]$ in each dimension. It scales exponentially with dimensionality.

Sparse Grid: The two major differences between the permutohedral lattice and the bilateral grid are the lattice used, and also the fact that the permutohedral lattice stores values sparsely in a hash table. In order to disambiguate these two effects, we constructed a sparse bilateral grid algorithm that uses the same hash table implementation. Similarly to the Gaussian KD-Tree and the permutohedral lattice, we do not allocate new lattice points during the blur stage. However, the time and memory complexity are still exponential in d, and indeed this technique is slower than the dense bilateral grid, while saving only a moderate amount of memory. This

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

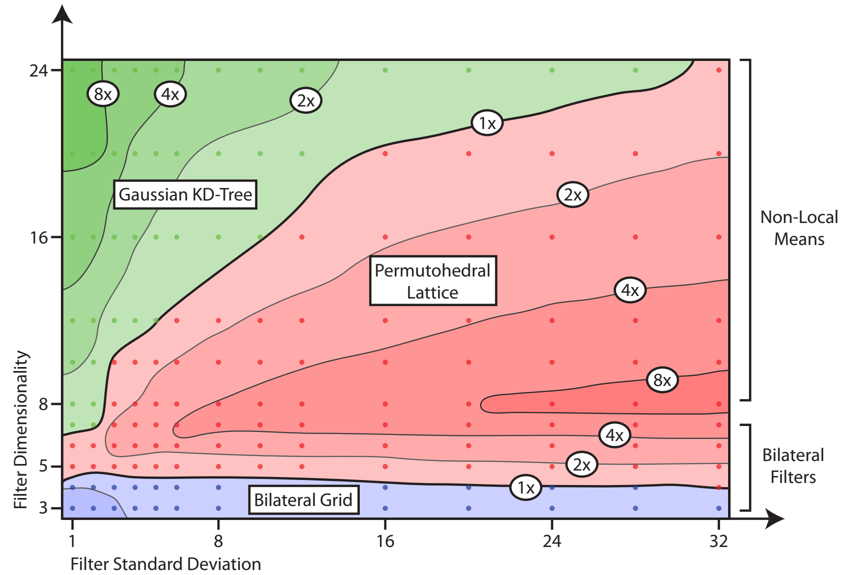

Figure 7: *This contour plot shows the fastest method for each dimensionality and spatial filter size, and how many times faster it is than the second fastest method. The bilateral grid [CPD07] is best for dimensionalities three and four. The Gaussian KD-Tree [AGDL09] is best for high dimensionalities and small filter sizes. The permutohedral lattice is the fastest method for dimensionalities from 5 (color bilateral filtering) up to around 20, depending on the filter size. Run times were sampled at the small dots and interpolated. Only methods capable of arbitrary-dimensional filters were compared, and the parameters of each method were tuned to achieve an PSNR relative to ground truth of roughly 45dB.*

shows that it is our choice of lattice, rather than our sparsity, which provides the advantage.

Gaussian KD-Tree: This is the Gaussian KD-Tree of Adams et al. [AGDL09]. Its run time is fairly constant across filter size and dimensionality, and its memory use is significantly lower than the other methods for dimensionalities 5 and up.

Improved Fast Gauss Transform: This is the fast multipole method of evaluating the Gauss Transform of C. Yang et al. [YDGD03]. It is a fully general method capable of extremely high accuracy, but even when tuned for speed, it is not particularly fast compared to the more approximate methods used in image filtering.

Real-Time $O(1)$ Bilateral Filtering: This is the method of Q. Yang et al. [YTA09] It is the fastest known method for grayscale bilateral filters ($d = 3$), but does not scale with dimensionality.

5. Interactive Applications

Images produced by applications of high-dimensional Gaussian filtering such as non-local means and the bilateral filter can depend heavily on the standard deviations chosen. The high speed and parallelizability of the permutohedral lattice make these filters easier to use by letting users interactively explore the parameter space. Our CUDA implementation of the lattice filters 800x600 images at 10 fps. We use this to implement the following interactive applications. Readers are encouraged to view the video accompanying this paper to see these applications demonstrated.

Interactive Color Bilateral Filtering The user is presented with sliders to control the standard deviations for each of the dimensions of the filter (x, y, r, g, b). The result of changing any of these values is presented immediately, making it easy to explore the space of bilateral filters.

Interactive Non-local Means As a preprocess, PCA is performed on the space of 7x7 patches centered about each pixel in the image, in order to reduce dimensionality without losing distance relationships. We use six PCA terms in addition to the two spatial terms as position dimensions. Six dimensions is the number suggested by Tasdizen [Tas08], who demonstrated that performing non-local means in this way produces superior results to non-local means without dimensionality reduction. Once again the user is presented with sliders to control the standard deviations, and the result is presented in real time, allowing for interactive-rate denoising.

Non-local Means Editing Non-local means can also be used for making edits to an image that propagate across similar colors, textures, or brightnesses, in the same manner as the propagating edits of [AP08]. By choosing appropriate position dimensions, local edits can be made to respect boundaries with respect to any set of local descriptors.

The user applies approximate edits with a few strokes. Each stroke paints values on a mask in those locations. The mask is then filtered with respect to the position dimensions. For our position dimensions, we use six PCA terms and two spatial terms, which captures changes in brightness, color, and texture. The filtered mask, which respects edges in the position dimensions, serves as an influence map for how the edit should be applied. For adjusting brightness, the user paints dark or bright values into the mask, and we simply multiply the input image by the filtered mask. The filtering is done at interactive rates, so the user can see the fully propagated edit as they apply their rough strokes.

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

6. Conclusions, Limitations, and Future Work

We have described the permutohedral lattice; a simplicial tessellation of $\mathbb{R}^d$ which results in the fastest known high-dimensional Gaussian filter for dimensionalities between 5 and 20. This covers many commonly used applications of the filter, such as color bilateral filtering and non-local means. Our method is consistently accurate, and has no parameters to tune other than the desired standard deviations. The algorithm is straightforward to implement, and a single-header-file implementation is included as supplemental material. The algorithm is also straightforward to parallelize to run on a GPU.

While we achieve high PSNRs with respect to ground truth (consistently within 45-50 dB), the barycentric interpolation we use is a linear interpolation, and the structure of the lattice becomes visible in the output for some filter settings. One could ameliorate this by storing higher order terms at each lattice point, such as Taylor series coefficients, in the manner of the improved fast Gauss transform [YDGD03]. Accuracy could also be increased by adopting a sampling strategy in addition to the interpolation.

Finally, while we have applied this lattice to the task of filtering, we expect that its properties will make it useful for any task which could benefit from a simplicial tessellation of high-dimensional space.

Acknowledgments

This work was partially funded by a Reed-Hodgson Stanford Graduate Fellowship. We would also like to thank Marc Levoy for his advice and support, as well as Nokia Research for their support.

References

[AGDL09] ADAMS A., GELFAND N., DOLSON J., LEVOY M.: Gaussian kd-trees for fast high-dimensional filtering. In *ACM Transactions on Graphics (Proc. SIGGRAPH 09)* (2009), pp. 1–12.

[AP08] AN X., PELLACINI F.: AppProp: all-pairs appearance-space edit propagation. In *ACM Transactions on Graphics (Proc. SIGGRAPH 08)* (2008), pp. 1–9.

[AW95] AURICH V., WEULE J.: Non-linear Gaussian filters performing edge preserving diffusion. In *Mustererkennung 1995, 17. DAGM-Symposium* (1995), pp. 538–545.

[BA09] BAEK J., ADAMS A.: *Some Useful Properties of the Permutohedral Lattice for Gaussian Filtering*. Tech. rep., Stanford University, 2009.

[BCM05] BUADES A., COLL B., MOREL J.-M.: A non-local algorithm for image denoising. In *Proc. CVPR* (2005), pp. 60–65 vol. 2.

[BCM08] BUADES A., COLL B., MOREL J.-M.: Nonlocal image and movie denoising. *International Journal of Computer Vision 76*, 2 (2008).

[BKC08] BROX T., KLEINSCHMIDT O., CREMERS D.: Efficient nonlocal means for denoising of textural patterns. *Image Processing, IEEE Transactions on 17*, 7 (July 2008), 1083–1092.

[BM05] BENNETT E. P., MCMILLAN L.: Video enhancement using per-pixel virtual exposures. In *ACM Transactions on Graphics (Proc. SIGGRAPH 05)* (2005), pp. 845–852.

[BPD06] BAE S., PARIS S., DURAND F.: Two-scale tone management for photographic look. In *ACM Transactions on Graphics (Proc. SIGGRAPH 06)* (2006), pp. 637–645.

[Buc07] BUCK I.: GPU computing: Programming a massively parallel processor. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), IEEE Computer Society, p. 17.

[CPD07] CHEN J., PARIS S., DURAND F.: Real-time edge-aware image processing with the bilateral grid. In *ACM Transactions on Graphics (Proc. SIGGRAPH 07)* (2007), p. 103.

[CS99] CONWAY J., SLOANE N.: *Sphere Packings, Lattice and Groups*, 3rd ed. Springer-Verlag, 1999.

[DCC*08] DARBON J., CUNHA A., CHAN T. F., OSHER S., JENSEN G. J.: Fast nonlocal filtering applied to electron cryomicroscopy. In *Proc. ISBI* (2008), pp. 1331–1334.

[DD02] DURAND F., DORSEY J.: Fast bilateral filtering for the display of high-dynamic-range images. In *Proc. SIGGRAPH 02* (2002), pp. 257–266.

[ED04] EISEMANN E., DURAND F.: Flash photography enhancement via intrinsic relighting. In *ACM Transactions on Graphics (Proc. SIGGRAPH 04)* (2004), pp. 673–678.

[KCLU07] KOPF J., COHEN M. F., LISCHINSKI D., UYTTENDAELE M.: Joint bilateral upsampling. In *ACM Transactions on Graphics (Proc. SIGGRAPH 07)* (2007), p. 96.

[MS05] MAHMOUDI M., SAPIRO G.: Fast image and video denoising via nonlocal means of similar neighborhoods. *IEEE Signal Processing Letters* (2005), 839–842.

[PD06] PARIS S., DURAND F.: A fast approximation of the bilateral filter using a signal processing approach. In *Proc. ECCV* (2006), pp. 568–580.

[PD09] PARIS S., DURAND F.: A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision 81* (2009), 24–52.

[Por08] PORIKLI F.: Constant time O(1) bilateral filtering. In *Proc CVPR 08* (2008), pp. 1–8.

[PSA*04] PETSCHNIGG G., SZELISKI R., AGRAWALA M., COHEN M., HOPPE H., TOYAMA K.: Digital photography with flash and no-flash image pairs. In *ACM Transactions on Graphics (Proc. SIGGRAPH 04)* (2004), pp. 664–672.

[SB97] SMITH S., BRADY J. M.: SUSAN: A new approach to low level image processing. *International Journal of Computer Vision 23* (1997), 45–78.

[Tas08] TASDIZEN T.: Principal components for non-local means denoising. In *Proc. ICIP* (2008), pp. 1728–1731.

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *Proc. ICCV* (1998), p. 839.

[WGY*06] WANG J., GUO Y., YING Y., LIU Y., PENG Q.: Fast non-local algorithm for image denoising. In *Proc. ICIP* (2006), pp. 1522–4880.

[WMM*04] WEBER M., MILCH M., MYSZKOWSKI K., DMITRIEV K., ROKITA P., SEIDEL H.-P.: Spatio-temporal photon density estimation using bilateral filtering. In *Computer Graphics International (CGI 2004)* (2004), IEEE, pp. 120–127.

[YDGD03] YANG C., DURAISWAMI R., GUMEROV N. A., DAVIS L.: Improved fast Gauss transform and efficient kernel density estimation. In *Proc. ICCV* (2003), pp. 664–671 vol.1.

[YTA09] YANG Q., TAN K.-H., AHUJA N.: Real-time O(1) bilateral filtering. In *Proc. CVPR* (2009), pp. 557–564.

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.

Appendix A: Properties of the Permutohedral Lattice

Proposition A.1 Given $\vec{x} \in H_d$, the following two statements are equivalent:

(i). The closest remainder-0 point to $\vec{x}$ is the origin.
(ii). $\max_k x_k - \min_k x_k \le d+1$.

Proof The closest remainder-0 point has the form $(d+1)\vec{z}$ for some $\vec{z} \in \mathbb{Z}^{d+1}$. Fix two distinct indices $i, j \in \{0, \ldots, d\}$ and define $\vec{z'}$ where

$$z'_k := \begin{cases} z_k + 1, & k = i, \\ z_k - 1, & k = j, \\ z_k, & \text{otherwise}. \end{cases}$$

By choice of $\vec{z}$, it must be that $(d+1)\vec{z}$ is closer to $\vec{x}$ than is $(d+1)\vec{z'}$. Therefore,

$$\begin{aligned} 0 &\le \|(d+1)\vec{z'} - \vec{x}\|^2 - \|(d+1)\vec{z} - \vec{x}\|^2 \\ &= \sum_{k=0}^{d} (d+1)^2 ({z'_k}^2 - z_k^2) - 2(d+1)x_k(z'_k - z_k) \\ &= 2(d+1)^2(1 + z_i - z_j) - 2(d+1)(x_i - x_j) \\ &= 2(d+1)\left[(d+1)(1 + z_i - z_j) - (x_i - x_j)\right]. \end{aligned}$$

Dividing both sides of the last inequality by $2(d+1)$ and rearranging the terms, we obtain

$$x_i - x_j \le (d+1)(1 + z_i - z_j). \tag{1}$$

(i⇒ii) Condition (i) implies $\vec{z} = \vec{0}$. Then (1) becomes,

$$x_i - x_j \le d+1.$$

Since this holds for all i, j, we obtain $\max_i x_i - \min_i x_i \le d+1$ as desired.

(ii⇒i) Condition (ii) implies that for all i, j,

$$x_i - x_j \ge \min_k x_k - \max_k x_k \ge -(d+1).$$

Combined with (1), this implies $z_j - z_i \le 2$. Because $(d+1)\vec{z} \in H_d$, the components of $\vec{z}$ must sum to zero. Both conditions hold only if each component is -1, 0 or 1.

Suppose nonzero components exist, i.e. $z_i = -1, z_j = 1$. For these particular values of i, j, (1) must hold as equality, meaning that $\vec{z'}$ and $\vec{z}$ are equidistant from $\vec{x}$. Thus we can continue adding 1 to a negative component and -1 to a positive component, until we reach the origin, without altering the distance to $\vec{x}$. So the origin must be the closest remainder-0 point to $\vec{x}$, or at least tied for the closest. □

Proposition A.2 Let $\vec{x}$ be an arbitrary point inside the canonical simplex, and let $b_0, \ldots, b_d$ be its barycentric coordinates in the simplex, i.e.

$$\vec{x} = \sum_{k=0}^{d} b_k \vec{s}_k, \quad \text{and} \quad \sum_{k=0}^{d} b_k = 1.$$

Then,

$$b_k = \begin{cases} \dfrac{x_{d-k} - x_{d+1-k}}{d+1}, & k \ne 0, \\ 1 - \dfrac{x_0 - x_d}{d+1}, & k = 0. \end{cases}$$

Clearly these weights sum to 1.

Proof Because the vertices of a (non-degenerate) simplex span the underlying Euclidean space, a barycentric decomposition exists and is unique. Thus it suffices to show that the given weights do yield $\vec{x}$. Let $\vec{y} = \sum_{k=0}^{d} b_k \vec{s}_k$. Then,

$$\begin{aligned} y_j &= \sum_{k=0}^{d} b_k \cdot (s_k)_j \\ &= \left[\sum_{k=0}^{d-j} b_k k\right] + \left[\sum_{k=d-j+1}^{d} b_k(k - (d+1))\right] \\ &= \left[\sum_{k=0}^{d} b_k k\right] - \left[(d+1) \sum_{k=d-j+1}^{d} b_k\right] \\ &= \left[\left(\frac{x_{d-1} - x_d}{d+1}\right) + 2\left(\frac{x_{d-2} - x_{d-1}}{d+1}\right) + \cdots \right. \\ &\quad \left. + d\left(\frac{x_0 - x_1}{d+1}\right)\right] - \left[\sum_{k=d-j+1}^{d} x_{d-k} - x_{d+1-k}\right] \\ &= \frac{-x_d - x_{d-1} - \cdots - x_1 + d x_0}{d+1} - (x_0 - x_j) \\ &= \frac{-x_d - x_{d-1} - \cdots - x_1 - x_0}{d+1} + x_j \\ &= x_j, \end{aligned}$$

as desired. □

Proposition A.3 The variance of splatting is $\dfrac{d(d+1)^2}{12}$.

Proof Consider the splatting kernel for the lattice point at the origin. The weights are given by $b_0 = 1 - \dfrac{\max_i x_i - \min_i x_i}{d+1}$. On the other hand, consider flattening the hypercube $[0, d+1]^{d+1}$ onto H_d. For each $\vec{x} \in H_d$, the points which project onto $\vec{x}$ must have the form $\vec{x} + k\vec{1}$. Since $\vec{x} + k\vec{1}$ must fall in the hypercube, we have $\forall i,\ 0 \le x_i + k \le d+1$. Hence, $k \in [-\min_i x_i,\ d+1 - \max_i x_i]$. That indicates that the mass of points that are projected onto $\vec{x}$ is proportional to $d+1 - (\max_i x_i - \min_i x_i)$, which in turn is proportional to the splatting kernel above.

Therefore, the variance of the splatting kernel equals that of the flattened hypercube, weighted by density. One may compute this by integrating over the original hypercube the second moment of the projected point about the origin:

$$\text{Variance} = \frac{\int_{[0,d+1]^{d+1}} \left\| \vec{y} - \frac{\sum y_i}{d+1} \vec{1} \right\|^2 d\vec{y}}{\int_{[0,d+1]^{d+1}} d\vec{y}}.$$

A number of straightforward algebraic manipulations yields the desired expression. □

© 2009 The Author(s)
Journal compilation © 2009 The Eurographics Association and Blackwell Publishing Ltd.