Fast Median and Bilateral Filtering

Ben Weiss[†]
Shell & Slate Software Corp.

Abstract

Median filtering is a cornerstone of modern image processing and is used extensively in smoothing and de-noising applications. The fastest commercial implementations (e.g. in Adobe® Photoshop® CS2) exhibit $O(r)$ runtime in the radius of the filter, which limits their usefulness in realtime or resolution-independent contexts. We introduce a CPU-based, vectorizable $O(\log r)$ algorithm for median filtering, to our knowledge the most efficient yet developed. Our algorithm extends to images of any bit-depth, and can also be adapted to perform bilateral filtering. On 8-bit data our median filter outperforms Photoshop's implementation by up to a factor of fifty.

CR Categories: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems – Sorting and Searching; I.4.3 [Image Processing and Computer Vision]: Enhancement – Filtering ; D.2.8 [Software Engineering]: Metrics – Complexity Measures; E.1 [Data Structures]: Arrays

Keywords: median filtering, bilateral filtering, rank-order filtering, sorting, image processing, algorithms, histograms, data structures, complexity, SIMD, vector processing

1 Introduction

1.1 Median Filtering

The median filter was introduced by Tukey [1977], and over the years tremendous effort has gone into its optimization and refinement. It provides a mechanism for reducing image noise, while preserving edges more effectively than a linear smoothing filter. Many common image-processing techniques such as rank-order and morphological processing are variations on the basic median algorithm, and the filter can be used as a steppingstone to more sophisticated effects. However, due to existing algorithms' fundamental slowness, its practical use has typically been restricted to small kernel sizes and/or low-resolution images.

[†]ben@shellandslate.com

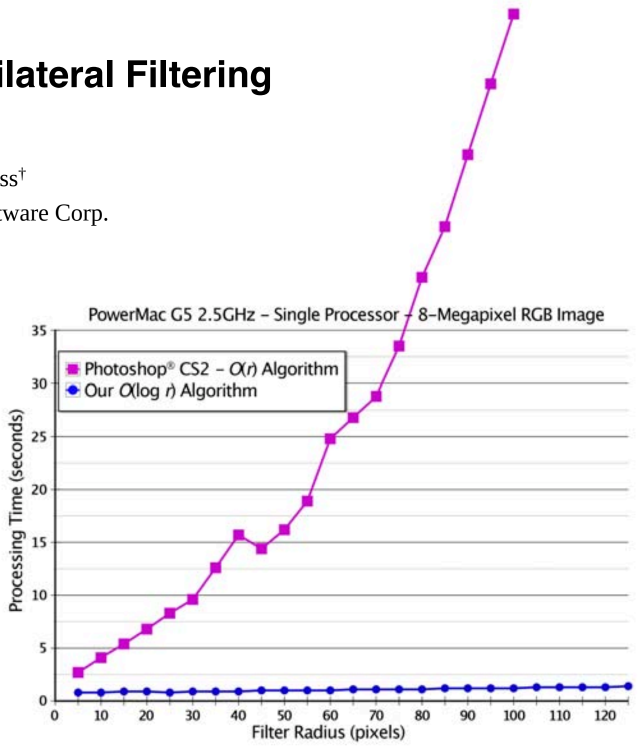


Figure 1: 8-Bit Median Filter Performance

Adobe® Photoshop® CS2 is the *de facto* standard for high-performance image processing, with a median filter that scales to radius 100. This filter exhibits roughly $O(r)$ runtime per pixel, a constraint which significantly reduces its performance for large filtering kernels. A variety of $O(r)$ algorithms are well known (e.g. Huang 1981), but it is not obvious that a faster algorithm should exist. The median filter is not separable, nor is it linear, and there is no iterative strategy for producing the final result, as there is with e.g. Gaussian Blur [Heckbert 1986], or the Fast Fourier Transform [Cooley et al. 1965]. A fast, high-radius implementation would be of considerable theoretical and practical value.

Gil et al. [1993] made significant progress with a tree-based $O(\log^2 r)$ median-filtering algorithm, but its per-pixel branching nature renders it ill-suited for deep-pipelined, vector-capable modern processors. Other efforts have resorted to massive parallelism on the presumption that a single processor is insufficient: according to Wu et al. [2003], "...designing a parallel algorithm to process [the median filter] is the only way to get a real-time response." Ranka et al. [1989] proposed a parallel algorithm with a processor-time complexity of $O(\log^4 r)$, but this curve actually scales worse than linear for $r < 55$ $(= e^4)$, the point at which a 1% increase in radius corresponds to a 1% increase in computation.

Our algorithm overcomes all of these limitations and achieves $O(\log r)$ runtime per pixel on 8-bit data, for both median and bilateral filtering. It is fully vectorizable and uses just $O(r)$ storage. It also adapts as an $O(\log^2 r)$ algorithm to arbitrary-depth images, on which it runs up to twenty times as fast as Photoshop's 16-bit Median filter. To our knowledge, the presented $O(\log r)$ algorithm is the most efficient 2D median filter yet developed, and processes 8-bit data up to fifty times faster than Photoshop's Median filter.

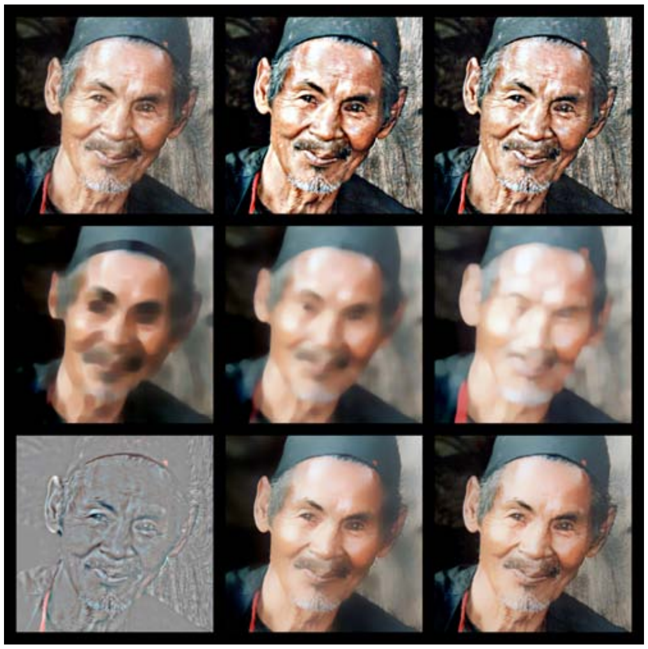


Figure 2: Median Filter Variations. Top row: original; sharpened with Gaussian; sharpened with median (note fewer halo artifacts.) Middle row: Filtered at 20th; 50th [median]; and 80th percentiles. Bottom row: "High Pass" using median; bilateral smoothing filter; logarithmic bilateral filter.

1.2 Bilateral Filtering

The Bilateral filter was introduced by Tomasi et al. [1998] as a non-iterative means of smoothing images while retaining edge detail. It involves a weighted convolution in which the weight for each pixel depends not only on its distance from the center pixel, but also its relative intensity. As described, the bilateral filter has nominal $O(r^2)$ computational cost per pixel. Photoshop® CS2's 16-bit Surface Blur filter reflects this $O(r^2)$ complexity, and becomes unusably slow for even moderate radii. On 8-bit data, Photoshop's Surface Blur exhibits a performance curve nearly identical to its 8-bit Median filter, suggesting that they share the same core $O(r)$ algorithm.

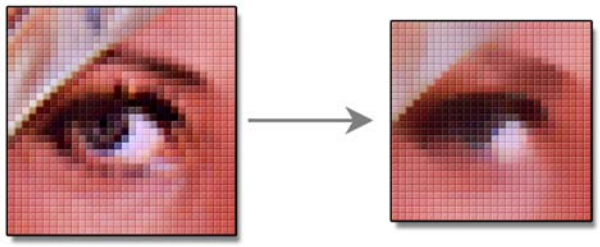
Durand et al. [2002] developed a much more efficient technique, refined and accelerated by Paris et al. [2006]. Durand's method approximates the bilateral by filtering subsampled copies of the image with discrete intensity kernels, and recombining the results using linear interpolation. It has the paradoxical property of becoming *faster* as the radius increases (due to greater subsampling), but also has some potential drawbacks. For one, it is not translation-invariant: the exact output is dependent on the phase of the subsampling grid. Also, the discretization may lead to a further loss of precision, particularly on high-dynamic-range images with narrow intensity-weighting functions.

Our bilateral filtering algorithm maintains high resolution in both space and intensity, and is translation-invariant. It is based on a box spatial kernel, which can be iterated to yield smooth spatial falloff. It is derived from the same core algorithm as our fast $O(\log r)$ median filter, and adapts to 16-bit and HDR data with minimal loss of precision.

1.3 Structure

Our approach in this paper will be first to illustrate the conventional $O(r)$ median algorithm for 8-bit images, and analyze its performance and limitations. Then we will show in steps how to improve it; first by constant factors, then into $O(\sqrt{r})$ and $O(\sqrt[3]{r})$ algorithms, and from there into an $O(\log r)$ algorithm. We will show how our approach adapts to higher bit-depth data, such as 16-bit and HDR floating-point. Finally, we will show how the algorithm can be adapted to perform bilateral filtering, and compare it with previous methods.

2 The Basic $O(r)$ Algorithm

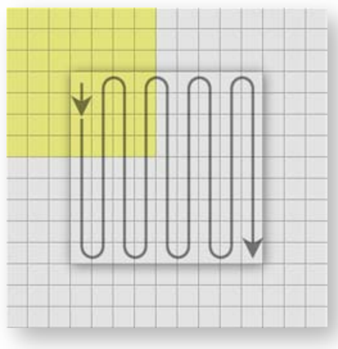


Consider the case of applying a radius-r median filter to an 8-bit image. Assume a source image that is larger than the destination by r pixels on all sides, to sidestep edge-related concerns. (In practice, we repeat edge pixels to fill undefined areas, and process color images on a per-channel basis.) Because the median filter is *local*, it can be applied to arbitrary-size images in tiles. As a consequence, its total runtime scales linearly with image area: $O(S^2)$ for an S-by-S image.

The fundamental property that concerns us here is *runtime per pixel*, as a function of filter radius. This corresponds to the performance a user will experience while adjusting the filter radius, and is the primary differentiating characteristic between median-filtering algorithms. For reference, a brute-force implementation can calculate each output pixel in $O(r^2 \log r)$ time, by sorting the corresponding $(2r + 1)^2$ -pixel input window and selecting the median value as output.

On discrete data, a radix-sort can be used to reduce the sorting complexity to $O(r^2)$ operations; this can be done for some floating-point data as well [Terdiman 2000]. In the case of 8-bit data, we use a 256-element histogram, H. Once the input values are added to H, the median value lies in the first index for which the sum of values to that index reaches $2r^2 + 2r + 1$. The median index can be found by integrating the histogram from one end until the appropriate sum is reached.

An improved algorithm was proposed by Huang [1981], based on the observation that adjacent windows overlap to a considerable extent. Huang's algorithm makes use of this sequential overlap to consolidate the redundant calculations, reducing the computational complexity to $O(r)$. A modified version of Huang's algorithm is below:



r: radius of median filter. (shown above as $r = 3$.)
H: 256-element histogram.
I: input image, $S + 2r$ pixels square.
O: output image [inset], S pixels square.

```
initialize H to I[0 .. 2r][0 .. 2r].  // yellow region
find median value m in H, write m to O[0][0].
   for row = 1 to S - 1:
      add values I[2r + row][0 .. 2r] to H.
      subtract values I[row - 1][0 .. 2r] from H.
      find median value m in H; write m to O[row][0].
   step sideways to next column (and process bottom to top, etc.).
```

Figure 3: Pseudocode for Huang's $O(r)$ Algorithm

Huang's algorithm is a significant improvement over the brute-force method. However, the window-sliding step dominates the calculation with $O(r)$ runtime per pixel, while the histogram-scanning takes constant time per pixel. This suggests that we should look for a way to make the window-sliding faster, even at the expense of making the histogram-scanning slower.

Observe that as the window zigzags through the image, it passes through each region several times, performing nearly the same operations on each pass. (Picture mowing your lawn back and forth, shifting sideways one centimeter each time.) This redundancy is considerable, and mirrors the adjacent-window overlap that led to Huang's algorithm.

The difficulty is that these redundant calculations occur at widely spaced time intervals in the computation; perhaps tens of thousands of processor cycles apart, so they cannot be combined using the same sequential logic that led to the $O(r)$ technique. Yet, eliminating these redundancies is the key to a dramatically faster algorithm.

3 The *O*(log *r*) Algorithm

3.1 Synchronicity

The fundamental idea behind this paper, and the mechanism that enables our fast algorithm, is the observation that *if multiple columns are processed at once*, the aforementioned redundant calculations *become sequential*. This gives us the opportunity to consolidate them, resulting in huge increases in performance.

3.2 Distributive Histograms

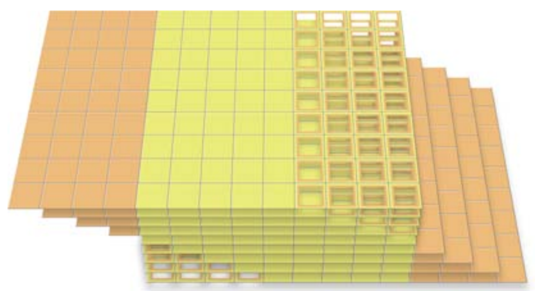
A straightforward adaptation of Huang's algorithm to process N columns at once involves the maintenance of N histograms, one per output column: H_0 .. H_{N-1}. This is essentially just a rearrangement of operations; the runtime complexity is unchanged. Each input pixel gets added to $2r + 1$ histograms over the course of filtering the image, leading to the $O(r)$ runtime complexity.

Fortunately, the explicit maintenance of each histogram H_n is unnecessary, due to the *distributive* property of histograms. This is where our approach diverges from Huang's algorithm. Histogram distributivity means that for disjoint image regions A and B:

$$H_{A \cup B}[v] \equiv H_A[v] + H_B[v] \tag{1}$$

In other words, if an image window W is the union of two disjoint regions A and B, then its histogram H_W is equal to $H_A + H_B$. The median element of W can then be found by scanning the *implicit* histogram H_W, splicing it together from H_A and H_B on the fly. (This extends to signed linear combinations; $H_A \equiv H_W - H_B$, etc.)

In the case of median-filtering N columns, our approach is to form a set H^* of *partial* histograms P_0 .. P_{N-1} (whose elements may be signed), such that each histogram H_0 .. H_{N-1} is representable as the *sum* of T partial histograms from H^*. Figure 4 shows how a row of pixels v_0 .. v_{2r+8} is added to H^*, for the case $N = 9$, $T = 2$.



Huang: H_0 .. H_8	Our Method: P_0 .. $P_8 \subseteq H^*$	Mapping:
$H_0[v_0 .. v_{2r}\ ]$++;	$P_0[v_0..v_3]$++; $P_0[v_{2r+1}..v_{2r+4}]$--;	$H_0 \equiv P_0 + P_4$
$H_1[v_1 .. v_{2r+1}]$++;	$P_1[v_1..v_3]$++; $P_1[v_{2r+2}..v_{2r+4}]$--;	$H_1 \equiv P_1 + P_4$
$H_2[v_2 .. v_{2r+2}]$++;	$P_2[v_2..v_3]$++; $P_2[v_{2r+3}..v_{2r+4}]$--;	$H_2 \equiv P_2 + P_4$
$H_3[v_3 .. v_{2r+3}]$++;	$P_3[\quad v_3]$++; $P_3[\quad v_{2r+4}]$--;	$H_3 \equiv P_3 + P_4$
$H_4[v_4 .. v_{2r+4}]$++;	$P_4[v_4$ .... $v_{2r+4}]$++;	$H_4 \equiv \quad P_4$
$H_5[v_5 .. v_{2r+5}]$++;	$P_5[v_4\quad]$--; $P_5[v_{2r+5}\quad]$++;	$H_5 \equiv P_5 + P_4$
$H_6[v_6 .. v_{2r+6}]$++;	$P_6[v_4..v_5]$--; $P_6[v_{2r+5}..v_{2r+6}]$++;	$H_6 \equiv P_6 + P_4$
$H_7[v_7 .. v_{2r+7}]$++;	$P_7[v_4..v_6]$--; $P_7[v_{2r+5}..v_{2r+7}]$++;	$H_7 \equiv P_7 + P_4$
$H_8[v_8 .. v_{2r+8}]$++;	$P_8[v_4..v_7]$--; $P_8[v_{2r+5}..v_{2r+8}]$++;	$H_8 \equiv P_8 + P_4$
$18r + 9$ operations	$2r + 41$ operations	$T = 2$

Figure 4: Adding a row of pixels to H^*, for the case $N = 9$, $T = 2$. Each layer shows how the corresponding histogram H_n is formed from partial histograms P_n in H^*. The pseudocode shows how a row of pixels v_0 .. v_{2r+8} is added to H^*. The "holes" represent pixels that are added to the central histogram P_4 but subtracted from partial histograms, canceling themselves out.

The histogram set H^* is arranged like a tree, with a central histogram (P_4) representing the input window for the central column, and the other partial histograms P_n representing the *difference* between the central and adjacent windows. The sum of each partial plus central histogram yields the full histogram for the corresponding square input window. By widening the yellow central region and fitting the partial histograms to its edges, the 9-column technique can be adapted to perform median filtering of arbitrary radius. The time spent modifying H^* is still $O(r)$, but with a much lower constant than Huang's algorithm. The median extraction time from H^* remains constant regardless of r.

The more fundamental improvement in efficiency comes when we allow the number of columns N to *vary* with r, conceptually adding more planes to Figure 4. For N output columns, the number of modifications to H^* per output pixel is $(N^2 + 4r + 1) / N$. (The graphic in Figure 4 show the case of $N = 9$, $r = 4$, requiring 98 adjustments to H^* per row or about 11 per output pixel.) Solving for N to minimize the number of adjustments gives $N \approx 2\sqrt{r}$, which yields $O(\sqrt{r})$ histogram modifications per pixel. Thus, the complexity of the $T = 2$, variable-N adaptive algorithm is $O(\sqrt{r})$.

3.3 Three Tiers and Beyond

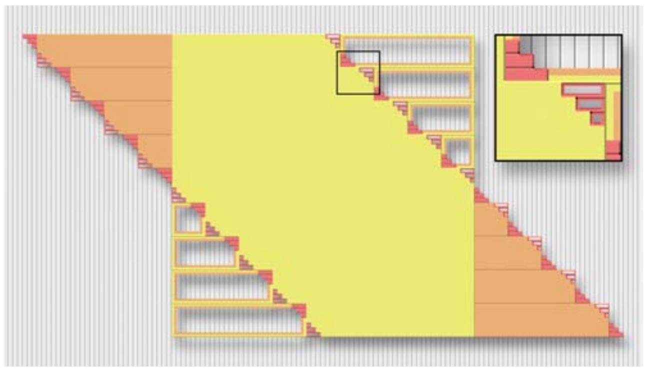


Figure 5: H^* Histogram Layout for $N = 63$, $T = 3$.

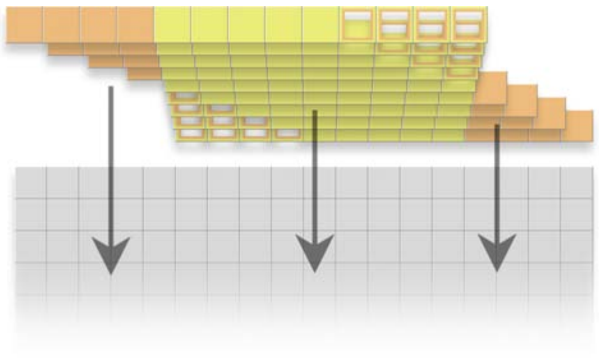
Figure 5 shows a layout for processing sixty-three columns at once. It is the three-tiered analogue of Figure 4, this time "viewed" from the side. There is a single shared histogram P_{31} [yellow] corresponding to the central window; eight partial histograms [orange] at seven-pixel intervals; and for each of these, six small partial histograms [red] at unit intervals; sixty-three histograms altogether. Each input pixel is added/subtracted to each histogram intersecting its column. In this example, a 63-by-1 block of output is produced at each iteration. The mapping of P_n to H_n becomes:

$$H_n = P_{31} + P_{7\lfloor n/7 \rfloor + 3} + P_n \qquad (2)$$

where the second and third terms are ignored if they match earlier terms (e.g., $H_{24} = P_{31} + P_{24}$.) The structure of H^* is recursive; the central yellow histogram forms a rough approximation to any particular H_n; the orange partial histograms refine that approximation, and the red histograms provide the final correction to make the sums exact. Once H^* is initialized, the full histogram of each of the 63 square input windows is expressible per Eq. 2 as the sum of one red histogram (or none), one orange histogram (or none), and the yellow central histogram. The illustrated case of $N = 63$, $T = 3$, $r = 31$ requires ~18 histogram modifications per output pixel. The median-extraction from H^* takes constant time, as the three partial histograms are spliced together on the fly.

For the general case of 3-tiered structures, processing N columns at once and with tier radix $\sqrt{N}$, the number of histogram adjustments per output pixel becomes $\sqrt{N} + ((4r + 2) / N)$. For radius r, solving for optimal N yields $N \approx 4r^{2/3}$, and the runtime of the three-tiered adaptive algorithm is therefore $O(\sqrt[3]{r})$.

In practice, three tiers covers the realistic range of implementation (into the hundreds), but our technique can be extended to arbitrary T. In the limit, a radius-r median filter can be computed across $N = O(r)$ columns at once, using N histograms arranged into $T = O(\log r)$ tiers of constant radix. For example, a radius *one-million* median filter can be computed across $N = 9^6 = 531{,}441$ columns at once, using 9^6 partial histograms arranged in seven tiers of radix 9, occupying roughly 500 megabytes of storage. Sliding the window from one row to the next requires $O(\log r) \approx 114$ histogram modifications per output pixel. Extracting each median takes $O(\log r)$ steps; in this case splicing up to seven partial histograms together to construct each H_n, counterbalancing the $O(\log r)$ complexity of writing to H^*. Therefore, the overall computational cost per pixel is $O(\log r)$. □



r: radius of median filter.
H^*: Array of partial histograms, processing N columns.
I: input image, $N + 2r$ pixels square.
O: output image, N pixels square.

```
for each row in [0 .. 2r]:      // Initialize H*
    Add row, I[row][0 .. 2r + N - 1] to H*, as per Figure 4
for each output pixel in O[0][col]:  // compute first N median values
    scan H_col (implicit in H*) to the find the median m,
    write O[0][col] = m.
for row = 1 to N - 1: // step from top to bottom of image
    add new bottom row, I[row + 2r][0 .. 2r + N - 1], to H*.
    subtract old top row, I[row - 1][0 .. 2r + N - 1], from H*.
    find N new median values in H*; write to O[row][0 .. N - 1].
```

Figure 6: Pseudocode for $O(\log r)$ Algorithm

3.4 Implementation Notes

Scanning the histogram from index zero to find the median takes about 128 steps on average. Huang [1981] suggested using each output value as a “pivot” to find the next median value: as H is scanned to find m, we keep track of the number of values $v < m$ in H. Then as we add and remove pixels from H, we keep a running count of how many values satisfy $v < m$. This allows us to scan the updated histogram starting from m, which is typically much faster than starting from index zero.

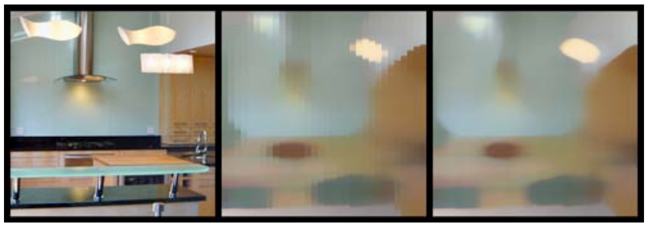


Figure 7: Pivot Tracking. The middle image shows the approximation obtained using one pivot per sixteen columns to track the smallest median values. H^* is then scanned upwards from these pivots (several columns at a time, vectorized) to yield the exact median result, right.

This heuristic adapts to the $O(\log r)$ algorithm by using $O(\log r)$ pivots across the N columns, with each pivot tracking the smallest median value in its respective columns. This approach obtains much of the benefit of the heuristic while preserving the $O(\log r)$ complexity. Since the pivot tracking involves many consecutive bytewise compares, it is ideally suited for vector optimization.

Finally, it is useful to *interleave* the partial histograms P_n in memory, so that multiple adjacent histograms can be modified simultaneously using vector loads and stores. This greatly accelerates the reading and writing of H^*.

4 Higher-Depth Median Filtering

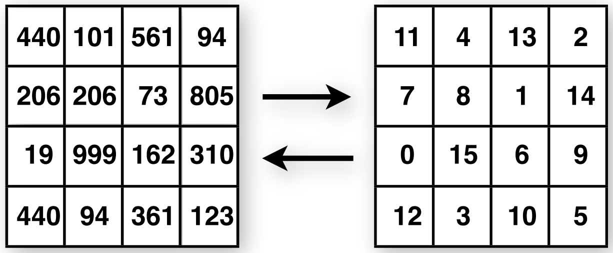
4.1 Adapting the 8-bit Algorithm

16-bit and HDR images have already become mainstream, so it is important that our median filter work with images of arbitrary bit-depth. A direct extension of the 8-bit algorithm is problematic, because the histograms must stretch to accommodate every possible value, growing exponentially with bit-depth. The algorithm still remains $O(\log r)$, but storage considerations render it impractical for 16-bit images and impossible for floating-point images.

4.2 The Ordinal Transform

H^* is reduced to a manageable size through a technique we call the *ordinal transform*. This involves *sorting* the input image values, storing the sorted list, and replacing each cardinal value with its ordinal equivalent. (Duplicate cardinal values map to consecutive ordinal values.) The median filter is then applied to the ordinal image, and the transform is inverted to restore the cardinal-valued result. The ordinal transform operates on images of any depth, in logarithmic or constant time per pixel.

In this operation, the nonlinearity of the median filter is crucial. Any linear filter (e.g., Gaussian blur) would not be invariant under the ordinal transform, but the median filter is! That is because rank-order is preserved; the k^{th}-smallest cardinal value maps to the k^{th}-smallest ordinal value. After the ordinal transform is applied, the median filtering proceeds as in Section 3, this time using single-bit histograms P_n (sufficient here because each ordinal value is unique in the image), and the results are inverse-transformed to yield the final filtered image.



440	101	561	94
206	206	73	805
19	999	162	310
440	94	361	123

11	4	13	2
7	8	1	14
0	15	6	9
12	3	10	5

Figure 8: The Ordinal Transform. Duplicate cardinal values (e.g. 94, 94, left) map to consecutive ordinal values (2 and 3, right).

Recall that the histogram elements in H^* can go negative. At first this appears problematic because the required range [-1, 0, 1] doesn’t fit into a single bit. However, since each summed implicit histogram value $H_n[v]$ can only be either zero or one, only the lowest bit from each partial histogram must participate in the summation. Hence a single bit is sufficient for each element of P_n, and the splicing accomplished through a bitwise XOR.

4.3 The Compound Histogram

For processing N columns in parallel, this approach still requires the allocation and maintenance of N single-bit histograms. However, due to the uniqueness of values in the ordinal image, we can take advantage of a much more efficient encoding.

Consider the full histogram obtained by splicing the n^{th} set of partial histograms in H^* (consisting of the central histogram plus one partial histogram from each tier), to yield the single-bit histogram for the n^{th} input window. Label this binary histogram B_n. By definition, the single bit $B_n[v]$ indicates whether the ordinal value v lies in the input window n.

Now, for $N <= \min(2r, 128)$, instead of allocating N binary histograms, we allocate a single *8-bit compound histogram* H^c. As rows of pixels $v = I[\text{row}][\text{col}]$ are added, we adjust H^c as follows:

$$H^c[v] = \begin{cases} \text{0xFF - col}, & \text{col} < N - 1 \\ \text{0x80}, & N - 1 <= \text{col} <= 2r \\ \text{0x80 - (col - }2r), & \text{col} > 2r \end{cases} \qquad (3)$$

Since the ordinal values in I can have any arrangement, the compound histogram H^c is filled in arbitrary order. As rows of pixels are removed, the corresponding elements of H^c are zeroed. The power of this technique becomes clear when it comes time to scan the implicit histogram B_n to find the n^{th} median output value.

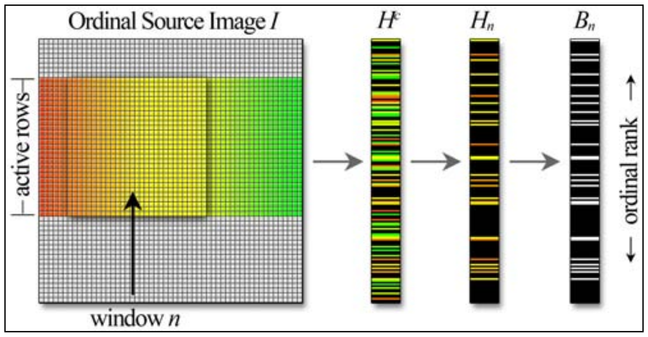


Figure 9: The Compound Histogram H^c

In our initial approach, each implicit histogram B_n was spliced together from $O(\log r)$ partial histograms, taking $O(\log r)$ time per element. With the compound histogram, using 8-bit modular arithmetic, elements of B_n can now be computed in *constant time:*

$$B_n[v] = (H^c[v] + n) >> 7. \quad (4)$$

For $N > 128$, this technique extends in a straightforward manner to 16-bit compound histograms, sufficient for $N <= 32768$, and so on. The computational complexity is independent of element size.

4.4 Coarse-To-Fine Recursion

There is one final detail. As the radius increases, the histogram size scales as $O(r^2)$, which directly affects the histogram scanning distance and thus the algorithm's time-complexity. This complication is addressed by computing the median in stages from coarse to fine precision. Alparone et al. [1994] applied a similar technique to the $O(r)$ algorithm, employing two levels of resolution to process 10, 12, or 14-bit images in faster (but still $O(r)$) time. Here we apply an analogous technique to our log-time algorithm.

In our case, the coarse-to-fine calculation is performed by right-shifting the ordinal image 8 bits at a time (or similar radix) until it reaches a fixed low resolution; e.g., 10 bits per pixel. Then the $O(\log r)$ algorithm from Section 3 is applied to the low-resolution data (whose values are no longer unique), storing not only the median values, but also the number of values *strictly below* the median. This result forms a pivot from which we calculate the median at the next-higher level of resolution. For example, if the lowest-resolution median value for a pixel is 0x84, and there are n values below 0x84 in its histogram, then there will be n values below 0x8400 in the next-higher-resolution histogram, and the median will be in [0x8400 .. 0x84FF]. This scanning is bounded by a constant [256] number of steps per iteration, with each iteration adding eight bits of precision to the output. The final iteration is performed using the compound histogram, which yields the full-precision ordinal result. The entire process requires $O(\log r)$ levels of recursion, each taking $O(\log r)$ time as shown in Section 3, for an overall computational complexity of $O(\log^2 r)$. □

4.5 Implementation Notes

Applying a radius-r median filter to an ordinal image cannot output any of the lowest $(2r^2 + 2r)$ ordinal values, because by definition the median must exceed that many values. The filter can thus treat all such values as a single low constant, and likewise the $(2r^2 + 2r)$ highest values as a single high constant, without affecting the final result. This "endpoint compression" can be incorporated into the ordinal transform, allowing input windows significantly larger than 2^{16} pixels to be filtered using 16-bit ordinal images.

Interestingly, since each ordinal value is unique, the median output for each pixel also tells us *where* in the source image that value came from, generating a vector field. On high-frequency images this field is quite noisy, but on smoother images it exhibits surprising structure. (Figure 14 on the last page is an emergent example of this structure.) Also, a variation of H^c where both row and column information is stored at each index can allow histogram elements of any computable region (e.g., a circle) to be determined in constant time. We have not fully explored these properties, but they suggest possible directions for future research.

For our implemented range of radii [1...127], the compound histogram is efficient enough not to require the coarse-to-fine recursion at all, except on carefully-constructed worst-case data. (Real-world images are invariably close to best-case.) In fact, the ordinal transform by itself is often the performance bottleneck. As shown in Figure 10, our implementation outperforms the 16-bit Median filter in Photoshop® CS2 by up to a factor of 20, with identical numerical results.

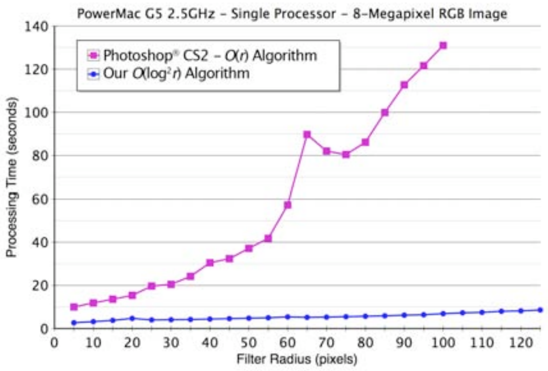


Figure 10: 16-Bit Median Filter Performance

5 The Bilateral Filter

The bilateral filter is a normalized convolution in which the weighting for each pixel p is determined by the spatial distance from the center pixel s, as well as its relative difference in intensity. In the literature (Tomasi et al. [1998] and Durand et al. [2002]), the spatial and intensity weighting functions f and g are typically Gaussian; Photoshop® CS2 implements a box spatial filter and triangular intensity filter. These functions multiply together to produce the weighting for each pixel. For input image I, output image J and window Ω, the bilateral is defined as follows:

$$J_s = \sum_{p\in\Omega} f(p-s)g(I_p - I_s)I_p \Big/ \sum_{p\in\Omega} f(p-s)g(I_p - I_s). \quad (5)$$

The special case of a spatial box-filter (with arbitrary intensity function) is worth studying, because the weighting function becomes *constant* for all pixels of a given intensity. Under this condition, the *histogram* of each spatial window becomes sufficient

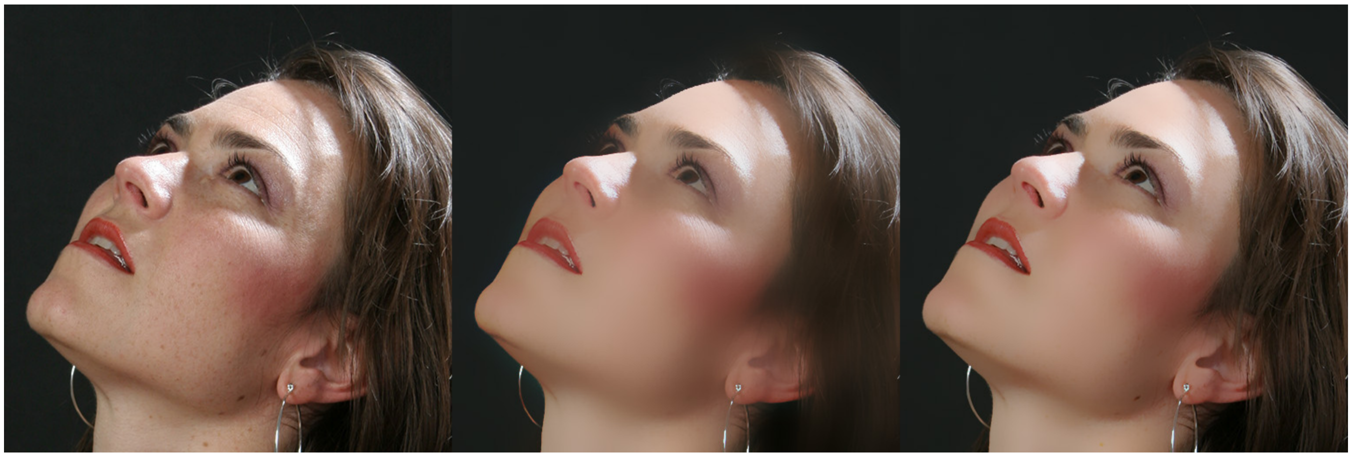


Figure 11: The Bilateral Filter. From left: 8-Bit Source Image; Linear-Intensity Bilateral (Eq. 5); Logarithmic-Intensity Bilateral (Eq. 6).

to perform the filtering operation. Our *O*(log *r*) median-filtering algorithm already generates these histograms, so the bilateral convolution can be appended in constant time per pixel, scaling with the support of the intensity function *g*.

For higher-precision data, one can either dither the source data into 8 bits before processing (which introduces surprisingly little error), or else downsample the source intensities into the histograms (along the lines of Paris et al. [2006]), which requires larger histogram elements but yields better accuracy. Durand et al. [2002] applied the bilateral to log-scaled images and re-expanded the result, but this approach can pose precision problems when filtering 8-bit data. Fortunately, this logarithmic approach can be approximated on linear data by scaling the width of *g* in proportion to the intensity of the center pixel while biasing the weight toward smaller values, yielding a new function *g*'. The rightmost image in Figure 11 shows the result of this logarithmic bilateral on 8-bit data, using a simple variable-width triangular function for *g*'. (Note the improved lip color and hair detail.) More sophisticated intensity functions can be precomputed for all (I_p, I_s). Our linear-data approximation to the logarithmic bilateral is as follows:

$$J_s = \sum_{p\in\Omega} f(p-s)g'(I_p/I_s)I_p \Big/ \sum_{p\in\Omega} f(p-s)g'(I_p/I_s). \quad (6)$$

where $$g'(x) = g(\log x)/\sqrt{x}. \quad (7)$$

One potential concern with our histogram-based method is the imperfect frequency response of the spatial box filter. Visual artifacts may resemble faint mach bands, but these artifacts tend to be drowned out by the signal of the preserved image (e.g., the images in Figure 11 are box-filtered.) Still, smooth spatial falloff is achievable with our method, using an iterative technique. Direct iteration of the bilateral can yield an unintentionally cartoonish look [Tomasi 1998], but *indirect* iteration is more effective. At each step the output is re-filtered, while continuing to use the *original* data for the intensity windows. For homogeneous areas or with wide intensity kernels, this converges to a Gaussian without creating the cartoonish look:

$$I_s^{n+1} = \sum_{p\in\Omega} f(p-s)g(I_p^n, I_s^0)I_p^n \Big/ \sum_{p\in\Omega} f(p-s)g(I_p^n, I_s^0). \quad (8)$$

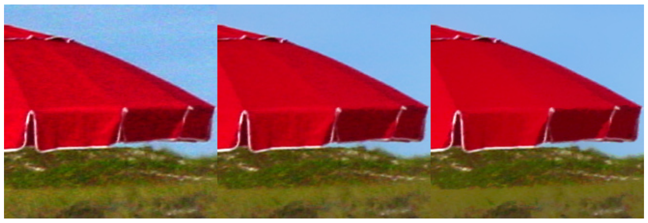


Figure 12: Original; One iteration; Three iterations (Eq. 8).

For the special case of the box-weighted bilateral, our technique achieves the discrete-segments result of Durand et al. [2002] in similar time, but with 256 segments instead of 10-20, and at full spatial resolution. This makes the result translation-invariant (avoiding artifacts due to the phase of the subsampling grid), and the high segment count allows high-dynamic-range images to be filtered with minimal loss of precision. Slight color artifacts may be introduced as a result of processing the image by channel, but we have found these also to be imperceptible on typical images.

With a single iteration and a fixed triangular intensity function (support 80 levels), our results numerically match Photoshop's Surface Blur output, with up to twenty-fold acceleration. The performance bottleneck (over 80% of the calculation) is the constant time spent multiplying each window's histogram by the intensity function, which accounts for the flatness of our performance curve. Reducing our implementation to 64 segments should nearly triple its speed, while maintaining very high quality results.

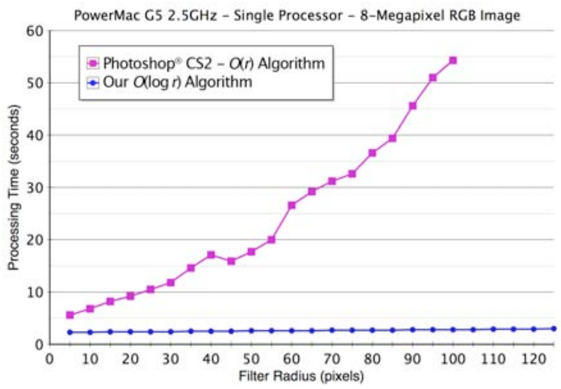


Figure 13: Bilateral Filter Performance

6 Conclusion

We have presented a logarithmic-time median filter algorithm, scalable to arbitrary radius and adaptable to images of any bit-depth. We believe this is the most efficient median algorithm yet developed, both in terms of theoretical complexity and real-world performance. Our algorithm can be extended to perform general rank-order filtering, and it is flexible enough to accomplish a wide variety of practical and creative tasks.

Significantly, we have shown that our algorithm can be adapted to perform bilateral filtering, where it becomes a highly effective noise-removal tool. Our algorithm provides a high-precision, translation-invariant, realtime implementation of the bilateral filter, and supports nonlinear intensity scaling, which greatly enhances the quality of the result.

Our algorithms have shown their advantage not only at high radii but across the spectrum. In the time it takes Photoshop® CS2 to process a 5x5 median or bilateral filter, our implementation can process any kernel up to 255x255. We have adapted our algorithm to multiple processors with near-linear performance gains, up to 3.2x faster on a four-processor system versus a single processor. The accompanying videos demonstrate the realtime performance of our median and bilateral filters.

Now that the speed of the median filter has been brought onto par with the workhorse filters of image-processing (e.g. Gaussian blur and FFT), we anticipate that the median filter and its derivatives will become a more widely used part of the standard image-processing repertoire. It is our hope that our algorithms spark renewed interest in this line of research, and we are confident that new applications and discoveries lie just around the corner.

Acknowledgments

Special thanks to Paul Heckbert for providing invaluable feedback in the early stages. Also thanks to Blaise Agüera y Arcas, Michael Herf, Klaus Schauser, Tobias Höllerer and Ian Gilman for their constructive critiques. To the talented Gretchen Elise for the use of her photo. To the reviewers for their time and insightful comments. Finally to Kai Krause, for the String Theory technique!

References

ALPARONE, L., CAPPELLINI, V., AND GARZELLI, A. 1994. A coarse-to-fine algorithm for fast median filtering of image data with a huge number of levels. Signal Processing, Vol. 39 No. 1-2, pp. 33-41.

COOLEY, J. H. AND TUKEY, J. 1965. An Algorithm for the Machine Calculation of the Complex Fourier series. Mathematics of Computation, vol. 19, pp. 297-301.

DURAND, F. AND DORSEY, J. 2002. Fast Bilateral Filtering for the Display of High-Dynamic-Range Images. ACM SIGGRAPH 2002.

HECKBERT, P. 1986. Filtering by Repeated Integration. ACM SIGGRAPH 1986.

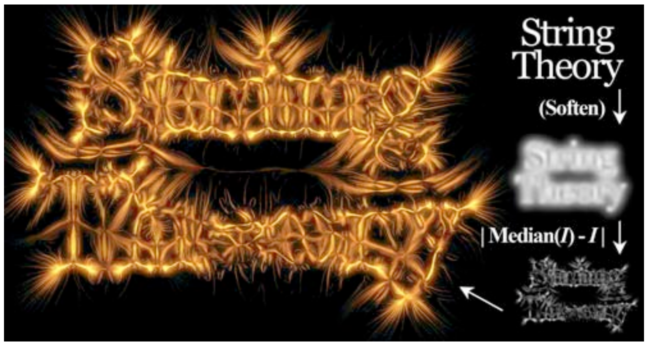


Figure 14: Kai's String Theory. A high-precision median filter is applied to a soft 16-bit mask, and the before-after ***difference*** (amplified 250x and colorized) is shown above. Far from the low-frequency result one might expect, the contours of the soft mask "beat" against the median filter's discrete sampling grid, producing an intricate filigree along rational field lines. This effect is ordinarily imperceptible, but with high amplification it lends itself to an unusual creative use of the median filter.

HUANG, T.S. 1981. *Two-Dimensional Signal Processing II: Transforms and Median Filters*. Berlin: Springer-Verlag, pp. 209-211.

GIL, J. AND WERMAN, M. 1993. Computing 2-D Min, Median, and Max Filters. IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 15 No. 5, pp. 504-507.

KABIR, I. 1996. *High Performance Computer Imaging.* Greenwich, CT. Manning Publications. pp. 181-192.

PARIS, S. AND DURAND, F. 2006. A Fast Approximation of the Bilateral Filter using a Signal Processing Approach. ECCV 2006.

PHA, T. Q. AND VLIET, L. J. V. 2005. Separable bilateral filtering for fast video preprocessing. IEEE Int. Conf. on Multimedia & Expo. CD1-4.

RANKA, S. AND SAHNI, S. 1989. Efficient Serial and Parallel Algorithms for Median Filtering. Proceeding 1989 International Conference on Parallel Processing, III-56 -- III-62.

TERDIMAN, P. 2000. Radix Sort Revisited. <http://www.codercorner.com /RadixSortRevisited.htm>

TANIMOTO, S. L. 1995. Fast Median Filtering Algorithms for Mesh Computers. Pattern Recognition, vol. 28, no. 12, pp. 1965-1972.

TOMASI , C. AND MANDUCHI , R. 1998. Bilateral filtering for gray and color images. In Proc. IEEE Int. Conf. on Computer Vision, 836–846.

TUKEY, J.W. 1977. *Exploratory Data Analysis*. Reading, MA. Addison-Wesley.

WEISS, B. 2006. Method and Apparatus for Processing Image Data. US Patent 7,010,163.

WU, C. H. AND HORNG, S. J. 2003. Fast and Scalable Selection Algorithms with Applications to Median Filtering. IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 10, pp. 983-992.