A Fast Approximation of the Bilateral Filter using the Discrete Fourier Transform

Pravin Nair[1], Anmol Popli[2], Kunal N. Chaudhury[1]

[1]Department of Electrical Engineering, Indian Institute of Science, Bangalore, India
(`nairpravin@ee.iisc.ernet.in,kunal@ee.iisc.ernet.in`)
[2]Department of Electrical Engineering, Indian Institute of Technology, Roorkee, India
(`anmal.uee2014@iitr.ac.in`)

Communicated by Charles Hessel   *Demo edited by* Charles Hessel

Abstract

The bilateral filter is a popular non-linear smoother that has applications in image processing, computer vision, and computational photography. The novelty of the filter is that a range kernel is used in tandem with a spatial kernel for performing edge-preserving smoothing, where both kernels are usually Gaussian. A direct implementation of the bilateral filter is computationally expensive, and several fast approximations have been proposed to address this problem. In particular, it was recently demonstrated in a series of papers that a fast and accurate approximation of the bilateral filter can be obtained by approximating the Gaussian range kernel using polynomials and trigonometric functions. By adopting some of the ideas from this line of work, we propose a fast algorithm based on the discrete Fourier transform of the samples of the range kernel. We develop a parallel C implementation of the resulting algorithm for Gaussian kernels, and analyze the effect of various extrinsic and intrinsic parameters on the approximation quality and the run time. A key component of the implementation are the recursive Gaussian filters of Deriche and Young.

Source Code

The ANSI C source code used in the demo can be downloaded from the web page of this article[1]. Compilation and usage instruction are included in the `README.txt` of the archive.

Keywords: bilateral filter; range kernel; discrete Fourier transform; fast algorithm; C code.

---

[1]https://doi.org/10.5201/ipol.2017.184

1   Introduction

The bilateral filter was proposed by Tomasi and Maduchi [12] as a non-linear extension of the classical Gaussian filter. It belongs to the class of edge-preserving filters that can smooth homogeneous regions and preserve edges at the same time. The bilateral filter has become a popular filtering tool in image processing, computer graphics, computer vision, and computational photography. We refer the interested reader to [9] for an account of the working of the filter and its various applications.

In this paper, we will consider grayscale images $f : \Omega \to \{0, 1, \dots, M\}$, where Ω is some finite rectangular lattice and M is the dynamic range. For example, $M = 255$ for an 8-bit grayscale image. The bilateral filtering of f is given by

$$\mathcal{B}[f](\boldsymbol{i}) = \frac{1}{\eta(\boldsymbol{i})} \sum_{\boldsymbol{j} \in [-\omega, \omega]^2} w(\boldsymbol{j}) \, g_{\sigma_r}(f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})) \, f(\boldsymbol{i} - \boldsymbol{j}), \tag{1}$$

where $\eta(\boldsymbol{i})$ is the normalization factor,

$$\eta(\boldsymbol{i}) = \sum_{\boldsymbol{j} \in [-\omega, \omega]^2} w(\boldsymbol{j}) \, g_{\sigma_r}(f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})). \tag{2}$$

As in the original formulation [12], we consider the spatial and range kernels to be Gaussian, namely,

$$w(\boldsymbol{i}) = \exp\left(-\frac{\|\boldsymbol{i}\|^2}{2\sigma_s^2}\right) \quad \text{and} \quad g_{\sigma_r}(t) = \exp\left(-\frac{t^2}{2\sigma_r^2}\right), \tag{3}$$

where $\|\cdot\|$ is the standard Euclidean norm. The parameters σ_s and σ_r are used to control the action of the spatial and range kernels. The window size is usually set to be $\omega = 3\sigma_s$ in practice. Similar to the classical Gaussian filter, the quantum of smoothing can be controlled using σ_s (the larger the value of σ_s, more is the smoothing). In contrast, the range kernel is fundamentally used to inhibit smoothing. For example, a small σ_r (narrow kernel) reduces the mixing of pixels from either sides of an edge, which helps in retaining the sharpness of the edge.

It is clear from (1) and (2) that one requires $\mathcal{O}(\omega^2)$ operations per pixel to compute the filter. This makes the real-time implementation of the bilateral filter challenging, especially when ω is large. In fact, one would expect ω to scale with the image size in most applications of the bilateral filter [9]. Several fast algorithms have been proposed that offer various trade-offs between speed and accuracy [6, 8, 10, 13, 4, 11, 3, 1, 2, 7]. We refer the interested reader to these recent papers [11, 3] for a survey and comparison of various fast algorithms.

The present work was motivated by a series of recent papers [4, 1, 2, 7], where it was demonstrated that a fast $\mathcal{O}(1)$ algorithm can be derived by approximating the Gaussian range kernel using polynomial and trigonometric functions. These functions have the so-called *shiftability* property [1] that can be used to derive an $\mathcal{O}(1)$ algorithm. At this point, we note that by an $\mathcal{O}(1)$ algorithm[2], we mean that the number of computations does not scale with the kernel width ω. It is a well-known fact that the Gaussian convolution,

$$\mathcal{G}[f](\boldsymbol{i}) = \sum_{\boldsymbol{j} \in [-\omega, \omega]^2} w(\boldsymbol{j}) f(\boldsymbol{i} - \boldsymbol{j}), \tag{4}$$

can be approximated using $\mathcal{O}(1)$ operations per pixel [5, 14]. This is achieved by first expressing (4) as a cascade of one-dimensional convolutions, and then a recursive formula is used for approximating each of the one-dimensional convolutions. In fact, as will be explained shortly, the basic idea behind the shiftable bilateral filter is to approximate (1) and (2) using a series of Gaussian convolutions, where the convolutions are performed on certain (pointwise) non-linear transforms of the input image.

---

[2]also referred to as a constant-time or linear-time algorithm in the image processing literature.

2 Proposed Algorithm

Similarly to [4, 11, 2, 7] the present idea is to use trigonometric sums for approximating the range kernel. The key difference is that instead of approximating the continuous kernel, we propose to approximate the discrete kernel samples. This was motivated by the recent work in [7], where the continuous kernel was first approximated using a Fourier series and then sampled for optimization purpose. We turn this idea[3] around and directly consider the sequence of kernel samples that appear in (1) and (2), which we then approximate using the discrete Fourier transform. To the best of our knowledge, the possibility of using the discrete Fourier transform has not been previously explored in this context. At this point, we would like to note that the proposed approach can be used to derive a fast algorithm for any generic range kernel and not just the Gaussian kernel, provided the discrete Fourier transform decays sufficiently fast.

Our starting point is the observation that the argument t in (3) assumes the pixel differences $f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})$ in (1) and (2). In particular, t takes values in $\Lambda_T = \{-T, \ldots, 0, \ldots, T\}$, where

$$T = \max \left\{ |f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})| : \boldsymbol{i} \in \Omega, \ \boldsymbol{j} \in [-\omega, \omega]^2 \right\}.$$

We will refer to T as the *local dynamic range*. It was observed in [2] that T is usually less than the full dynamic range M for natural images. A direct computation of T can be slow when ω is large. In fact, it has the same $\mathcal{O}(\omega^2)$ complexity as that of the bilateral filter. A fast recursive algorithm for computing T was proposed in [2], where the run-time does not depend on ω.

Now, consider the sequence of samples $g(-T), \ldots, g(0), \ldots, g(T)$, where

$$g(n) = \exp \left(-\frac{n^2}{2\sigma_r^2} \right) \qquad (n \in \Lambda_T). \tag{5}$$

There are a total of $|\Lambda_T| = 2T + 1$ samples. The discrete Fourier transform of (5) is given by

$$\hat{g}(k) = \frac{1}{|\Lambda_T|} \sum_{n \in \Lambda_T} g(n) \exp(-\iota \nu n k) \qquad (k \in \Lambda_T). \tag{6}$$

where $\iota = \sqrt{-1}$ and $\nu = 2\pi/|\Lambda_T|$. A well-known fact is that we can exactly reconstruct the samples from (6), namely, we have the inversion formula

$$g(n) = \sum_{k \in \Lambda_T} \hat{g}(k) \exp(\iota \nu k n) \qquad (n \in \Lambda_T). \tag{7}$$

The above representation is of interest for a couple of reasons. First, we are able to write the samples in terms of the shiftable exponential function. The algorithmic advantage that we can derive from the shiftable expansion in (7) will be made explicit shortly.

The second important point is that the Fourier transform of the (truncated) Gaussian (5) decays rapidly, provided that the truncation width T is sufficiently large compared to the effective support of the Gaussian (which is proportional to σ_r). In particular, we set

$$T_{\max} = \max(T, 3.2\sigma_r), \tag{8}$$

which is then used in place of T in (6) and (7). Notice that when σ_r is sufficiently small, $T_{\max} = T$. In this case, the width T is sufficiently large compared to the effective support of the Gaussian. Note that the smaller $3.2\sigma_r$ compared to the dynamic range, the wider the Fourier transform of the range

---

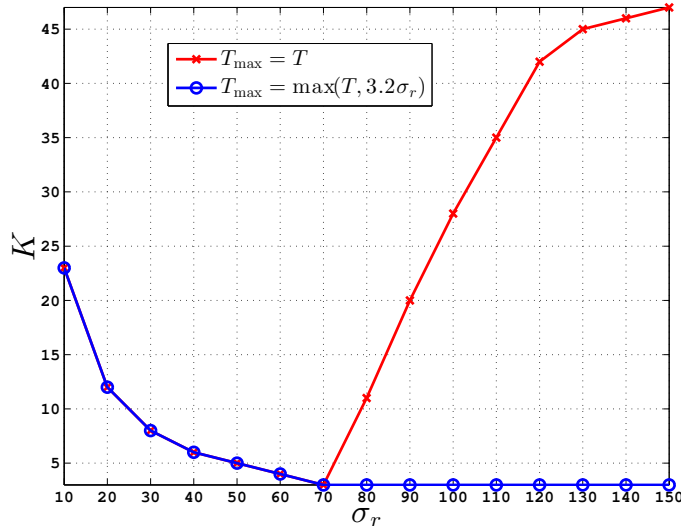[3]We wish to thank the reviewer for raising a question that led to this idea.

Figure 1: Demonstration of the advantage obtained using the proposed rule (8), where K was computed using (10). Notice that the $T_{\max} = T$ rule (that was used in [7]) results in larger values of K compared to the proposed rule when $\sigma_r > 70$. For this experiment, $\epsilon = 0.01$ and $T = 255$.

kernel. That is, the Fourier transform of the Gaussian that decays the most rapidly is obtained when $T_{\max} = 3.2\sigma_r$. However, we cannot take $T_{\max} < T$, since the kernel takes values in the interval $[-T, T]$. The lower bound on $T_{\max}$ is thus T. Hence the need to compute the local dynamic range instead of working with M. On the other hand, for large values of σ_r, i.e., when $T < 3.2\sigma_r$, we use $T_{\max} = 3.2\sigma_r$. This ensures that the samples in (5) are sufficiently concentrated over the domain $\Lambda_{T_{\max}}$ used in the Fourier transform. This produces a smoother transition at the boundary, which causes the Fourier coefficients to decay rapidly. In our case, we empirically found that the choice $3.2\sigma_r$ gives satisfactory results. In Figure 1, we demonstrate the utility of (8) over the $T_{\max} = T$ rule that was used in [7].

We propose to approximate (7) using partial sums corresponding to the largest Fourier coefficients. In particular, for some fixed approximation order $0 \leq K \leq T_{\max}$, we consider the K-term approximation

$$g_K(n) = \sum_{|k| \leq K} \hat{g}(k) \exp(\iota \nu k n) \qquad (n \in \Lambda_{T_{\max}}). \tag{9}$$

Clearly, $g_K(n)$ approaches $g(n)$ as K gets large, and $g_K(n) = g(n)$ in the extreme case when $K = T_{\max}$. The important point is that we can obtain a fairly accurate approximation when $K \ll T_{\max}$. In particular, given some user-defined tolerance $\epsilon > 0$, we pick the smallest K such that

$$\max \left\{ |g_K(n) - g(n)| : n \in \Lambda_{T_{\max}} \right\} \leq \epsilon. \tag{10}$$

The variation of K with ϵ for different values of σ_r is shown in Figure 2. Some of the reconstructions are shown in Figure 3.

After replacing the original range kernel with (9), we obtain the following approximations of (1) and (2)

$$\tilde{\mathcal{B}}[f](\boldsymbol{i}) = \frac{1}{\tilde{\eta}(\boldsymbol{i})} \sum_{\boldsymbol{j} \in [-\omega, \omega]^2} w(\boldsymbol{j}) g_K(f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})) f(\boldsymbol{i} - \boldsymbol{j}), \tag{11}$$

and

$$\tilde{\eta}(\boldsymbol{i}) = \sum_{\boldsymbol{j} \in [-\omega, \omega]^2} w(\boldsymbol{j}) g_K(f(\boldsymbol{i} - \boldsymbol{j}) - f(\boldsymbol{i})). \tag{12}$$
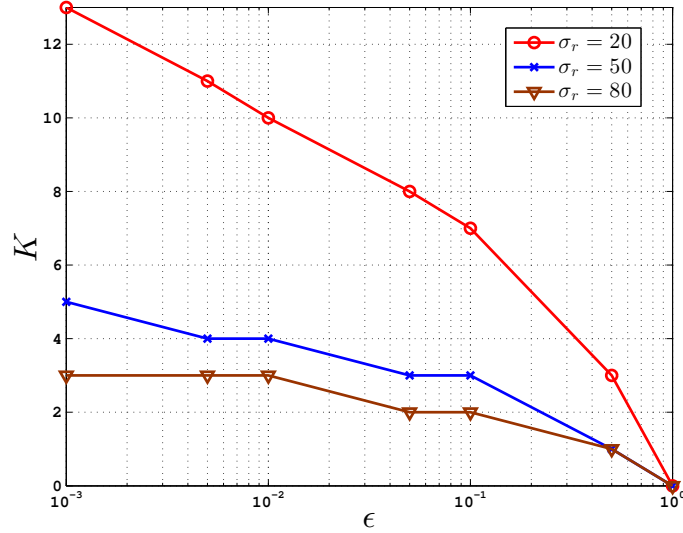
Figure 2: Variation of K with tolerance ϵ for three different values of σ_r ($T = 255$).

In particular, note that we can write the numerator of (11) as

$$\sum_{\boldsymbol{j}\in[-\omega,\omega]^2} w(\boldsymbol{j})\Big(\sum_{|k|\leq K} \hat{g}(k)\exp\big(\iota\nu k\left(f(\boldsymbol{i}-\boldsymbol{j})-f(\boldsymbol{i})\right)\big)\Big)f(\boldsymbol{i}-\boldsymbol{j}).$$

We next apply the addition-multiplication property $\exp(t+s) = \exp(t)\exp(s)$, and exchange the two sums, to arrive at

$$\sum_{|k|\leq K} \hat{g}(k)\exp(-\iota\nu k f(\boldsymbol{i})) \sum_{\boldsymbol{j}\in[-\omega,\omega]^2} w(\boldsymbol{j})\exp\big(\iota\nu k f(\boldsymbol{i}-\boldsymbol{j})\big)f(\boldsymbol{i}-\boldsymbol{j}).$$

Notice that the inner summation is simply a Gaussian convolution of the form in (4). In particular, we can write the numerator in (11) as

$$\sum_{|k|\leq K} \hat{g}(k)F_k(\boldsymbol{i})\mathcal{G}[H_k](\boldsymbol{i}), \tag{13}$$

where the images $F_k : \Omega \to \mathbb{C}$ and $H_k : \Omega \to \mathbb{C}$ are given by

$$F_k(\boldsymbol{i}) = \exp(-\iota\nu k f(\boldsymbol{i})) \quad \text{and} \quad H_k(\boldsymbol{i}) = f(\boldsymbol{i})F_k^*(\boldsymbol{i}) \qquad (\boldsymbol{i}\in\Omega).$$

It is now easy to see that (12) can be written as

$$\tilde{\eta}(\boldsymbol{i}) = \sum_{|k|\leq K} \hat{g}(k)F_k(\boldsymbol{i})\mathcal{G}[F_k^*](\boldsymbol{i}). \tag{14}$$

Notice that we need to perform a series of Gaussian convolutions to compute (13) and (14). Moreover, notice that

$$F_{-k} = F_k^*, \quad \mathcal{G}[H_{-k}] = \mathcal{G}[H_k]^*, \quad \text{and} \quad \mathcal{G}[F_{-k}^*] = \mathcal{G}[F_k^*]^*.$$

In other words, it is not necessary to compute the convolutions corresponding to the negative indices. In fact, we can write (13) and (14) as[4]

$$\hat{g}(0)\mathcal{G}[f](\boldsymbol{i}) + 2\cdot\text{real}\left\{\sum_{k=1}^{K} \hat{g}(k)F_k(\boldsymbol{i})\mathcal{G}[H_k](\boldsymbol{i})\right\},$$

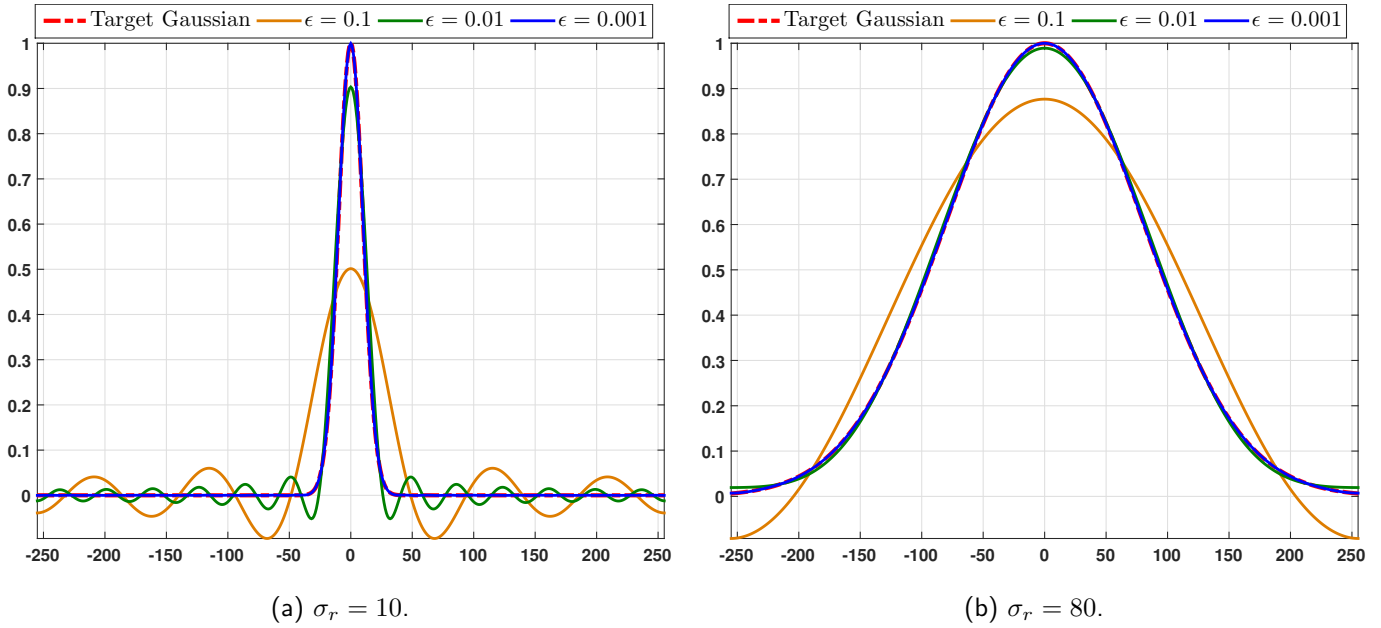[4]real(z) and z^* denote the real part and the complex conjugate of $z \in \mathbb{C}$.

(a) $\sigma_r = 10$.   (b) $\sigma_r = 80$.

Figure 3: Approximation of the Gaussian range kernel over $[-255, 255]$ for various values of ϵ. For a fixed ϵ, we computed K using (10).

and

$$\hat{g}(0)\mathcal{G}[F_0^*](\boldsymbol{i}) + 2 \cdot \mathrm{real} \left\{ \sum_{k=1}^{K} \hat{g}(k) F_k(\boldsymbol{i}) \mathcal{G}[F_k^*](\boldsymbol{i}) \right\}.$$

Notice that a total of $2(K+1)$ convolutions are required to compute (13) and (14). As mentioned earlier, each of the Gaussian convolutions can be computed using $\mathcal{O}(1)$ operations per pixel [5, 14], and hence the overall complexity of computing (11) is $\mathcal{O}(1)$ per pixel. The steps of the algorithm are described in Algorithm 1. In the description, we have used $\oplus, \otimes$ and $\oslash$ to denote pointwise addition, multiplication, and division of two images.

3   Implementation

We now discuss the $\mathcal{O}(1)$ implementation of the Gaussian convolution in (4) and the parallelization of Algorithm 1. For the Gaussian convolution, we experimented with various approximations and selected the proposals of Deriche [5] and Young [14] which appear to offer the best trade-off between speed and accuracy.

3.1   Gaussian Convolution

As it is well-known, we can efficiently compute (4) using the *separability* of the spatial Gaussian kernel. In particular, consider the one-dimensional convolution

$$\mathscr{G}[f](i) = \sum_{j \in [-\omega, \omega]} \exp\left(-\frac{j^2}{2\sigma_s^2}\right) f(i-j). \tag{15}$$

We can compute (4) by applying (15) along the rows of the input image, and then applying (15) along the columns of the intermediate image. Using separability, we can already cut down the number

---

Algorithm 1: Fast Bilateral Filter.

    Data: Image $f : \Omega \rightarrow \{0, 1, \ldots, M\}$;
    Parameters: Filter parameters σ_s and σ_r, and tolerance ϵ;
    Result: Output given by (11);
    // *Internal parameters*
    Compute T for the input image using the algorithm in [2];
    $T_{\max} \leftarrow \max(T, 3.2\sigma_r)$;
    $\nu \leftarrow 2\pi/(2T_{\max} + 1)$;
    Based on ϵ, compute K and $\{\hat{g}(k) : k = 0, 1, \ldots, K\}$ using the rule in (10);
    // *Initialization*
    $P \leftarrow \hat{g}(0)\mathcal{G}[f]$;
    $Q \leftarrow \hat{g}(0)\mathcal{G}[F_0^*]$;
    // *Gaussian convolutions*
    for $k = 1, \ldots, K$ **do**
        $F \leftarrow \exp(-\iota\nu k f)$;
        $P \leftarrow P \oplus 2 \cdot \text{real}(\hat{g}(k)\,(F \otimes \mathcal{G}[f \otimes F^*]))$;
        $Q \leftarrow Q \oplus 2 \cdot \text{real}(\hat{g}(k)\,(F \otimes \mathcal{G}[F^*]))$;
    // *Output*
    $\tilde{\mathcal{B}}[f] \leftarrow P \oslash Q$.

---

of operations per pixel from $\mathcal{O}(\omega^2)$ to $\mathcal{O}(\omega)$. This can be reduced to $\mathcal{O}(1)$ by approximating the underlying continuous kernel in (15), namely,

$$g(x) = \exp\left(-\frac{x^2}{2\sigma_s^2}\right). \tag{16}$$

In particular, we have implemented and compared the following approximations: Deriche [5], Young [14], and repeated box convolutions [14]. We have compared these with the exact FFT implementation[5] of (15) for different values of σ_S. A typical comparison is shown in Figure 4. We notice that Young's approximation offers better trade-off between accuracy and speed (for Algorithm 1) when σ_r is small. On the other hand, Deriche's approximation works better when σ_r is large. We experimentally found that Deriche's approximation has an edge (in terms of the trade-off between accuracy and speed) over Young's approximation when $T_{max} \leq 3.5\sigma_r$. We have used this rule in the C code.

    For completeness, we now provide brief descriptions of the schematic behind the approximations in [5, 14]; we refer the reader to the original papers for further details.

    Deriche's approximation. In [5], Deriche used a sum of weighted exponentials with complex coefficients and exponents to approximate (16). The filter order (the number of exponentials) dictates the accuracy and run-time of the approximation. For our implementation, we have used the following third-order approximation [5]

$$g_a(x) = 1.898\exp\left(-1.556\frac{x}{\sigma_s}\right) - \left(0.8929\cos\left(1.475\frac{x}{\sigma_s}\right) - 1.021\sin\left(1.475\frac{x}{\sigma_s}\right)\right)\exp\left(-1.512\frac{x}{\sigma_s}\right). \tag{17}$$

    The normalized mean square error between the lattice samples of (16) and (17) is $\sim$ 6e-06 [5]. In particular, let (h_k) be lattice samples of (17). Deriche's algorithm is based on splitting (h_k) into causal and anti-causal components, $h_k = h_k^+ + h_k^-$, where the transfer functions of h_k^+ and h_k^- are

$$H^+(z^{-1}) = \frac{a^+ + b^+ z^{-1} + c^+ z^{-2}}{1 - p^+ z^{-1} - q^+ z^{-2} - r^+ z^{-3}}, \tag{18}$$

---

[5]This FFT package was used: https://sourceforge.net/p/kissfft/discussion/?source=navbar.
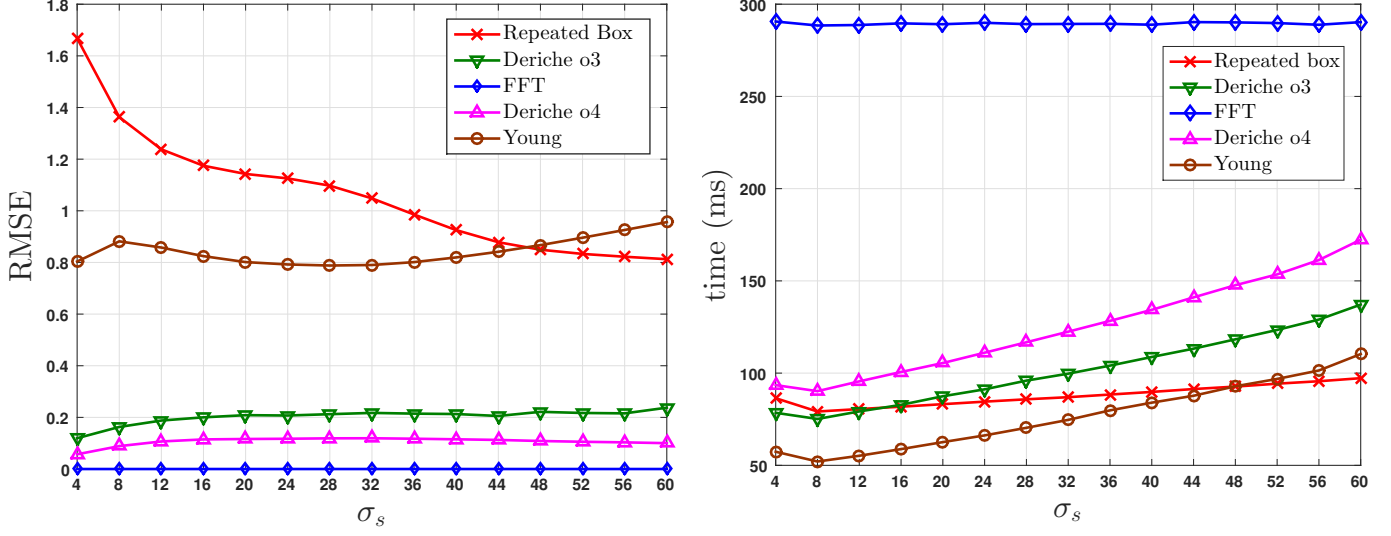
Figure 4: Comparison of the accuracy and speed of different approximations of (4). The experiments were performed on lena.png. The root-mean-square error (RMSE) is between (4) and the proposed approximation. The precise definition of RMSE is provided in (23).

Algorithm 2: Deriche's Approximation (third order).

Data: Sequence $f(0), \dots, f(N-1)$;

Parameter: σ_s;

Result: Approximation of (15);

Compute $a^+, b^+, c^+, p^+, q^+, r^+, a^-, b^-, c^-, p^-, q^-,$ and r^- in (18) and (19) using σ_s;

// *Compute causal part*

for $k = 0, \dots, N-1$ **do**

    $\mathscr{G}^+(k) = c^+ f(k-2) + b^+ f(k-1) + a^+ f(k) + r^+ \mathscr{G}^+(k-3) + q^+ \mathscr{G}^+(k-2) + p^+ \mathscr{G}^+(k-1)$;

// *Compute anti-causal part*

for $k = N-1, \dots, 0$ **do**

    $\mathscr{G}^-(k) = a^- f(k+1) + b^- f(k+2) + c^- f(k+3) + p^- \mathscr{G}^-(k+1) + q^- \mathscr{G}^-(k+2) + r^- \mathscr{G}^-(k+3)$;

// *Output*

for $k = 0, \dots, N-1$ **do**

    $\mathscr{G}[f](k) = \mathscr{G}^+(k) + \mathscr{G}^-(k)$

---

and

$$H^-(z) = \frac{a^- z^1 + b^- z^2 + c^- z^3}{1 - p^- z^1 - q^- z^2 - r^- z^3}. \tag{19}$$

The formula for the coefficients in (18) and (19) are described in equations (21)-(22) and (31) in [5]. The implementation of the causal and anti-causal filters are described in Algorithm 2.

Algorithm 3: Young's Approximation.

Data: Sequence $f(0), \ldots, f(N-1)$;
Parameter: σ_s;
Result: Approximation of (15);
Compute $b_0^+, b_1^+, b_2^+, b_0^-, b_1^-, b_2^-$, and b_{in} in (20) and (21) using σ_s;
// *Compute causal part*
for $k = 0, \ldots, N-1$ **do**
    $\mathscr{G}^+(k) = b_{\text{in}} f(k) + b_0^+ \mathscr{G}^+(k-3) + b_1^+ \mathscr{G}^+(k-2) + b_2^+ \mathscr{G}^+(k-1)$;
// *Output*
for $k = N-1, \ldots, 0$ **do**
    $\mathscr{G}[f](k) = b_{\text{in}} \mathscr{G}^+(k) + b_0^- \mathscr{G}[f](k+1) + b_1^- \mathscr{G}[f](k+2) + b_2^- \mathscr{G}[f](k+3)$;

Young's approximation. This is based on approximating (16) in the transform domain using a rational form [14]. More specifically, the Laplace transform of the Gaussian is approximated using

$$\mathscr{G}(s) = \frac{A_0}{a_0 - (a_2 q^2)s^2 + (a_4 q^4)s^4 - (a_6 q^6)s^6}.$$

This is next factored as $\mathscr{G}(s) = \mathscr{G}^+(s)\mathscr{G}^-(s)$, where $\mathscr{G}^+(s)$ has poles in left half plane (causal) and $\mathscr{G}^-(s)$ has poles in right half plane (anti-causal). The causal and anti-causal components are mapped into stable difference equations with transfer functions

$$H^+(z) = \frac{b_{\text{in}}}{1 - b_2^+ z^{-1} - b_1^+ z^{-2} - b_0^+ z^{-3}}, \tag{20}$$

and

$$H^-(z) = \frac{b_{\text{in}}}{1 - b_0^- z^1 - b_1^- z^2 - b_2^- z^3}. \tag{21}$$

The coefficients in (20) and (21) are polynomial functions of q. The exact formula is given by equation (11b) in [14]. The formula for the coefficients in (20) and (21) are provided in equations (8a), (8b), (8c), and (10) in [14]. The difference equations corresponding to (20) and (21) are implemented recursively. The complete procedure is summarized in Algorithm 3.

3.2 Parallelization

We now discuss the parallel implementation of Algorithm 1. A straightforward parallelization can be achieved by executing the "for-loop" in Algorithm 1 in parallel on separate cores, and accumulating the output of each core in a shared memory location. We have performed multi-threading in C using the OpenMP API. The description of the parallelized algorithm is provided in Algorithm 4. The main steps are as follows:

1. We extract the number of physical cores n_C from the system information.

2. Based on n_C, we fix the number of threads n_T to be spawned by the master thread.

3. Parallel regions with n_T threads are created.

4. Each thread is mapped to a distinct physical core.

Since the operating system assigns at most two virtual cores to a distinct physical core, n_T is at most twice the number of physical cores. A master thread sets up parallel regions using the "#pragma omp parallel" preprocessor directive. By default, n_T is equal to the number of hyperthreads on the system. Every thread uses a part of the memory to store temporary data and the processed output. Hence, the memory consumption increases with n_T. However, if n_T is reduced, then the possibility of two different hyperthreads being assigned to the same physical core increases. This would not produce the best performance as some cores would remain idle. Thus, an optimal trade-off between memory consumption and the number of threads spawned has to be struck. We have set the limit on n_T to be 8 (cf. line 5 of Algorithm 4) to limit the memory consumption and optimize the overall performance.

Note that the images $F_0, \ldots, F_K$ in Algorithm 1 can be efficiently computed by setting

$$F_n(\boldsymbol{i}) = F_{n-1}(\boldsymbol{i}) \cdot F_1(\boldsymbol{i}) \qquad (n \geq 1), \tag{22}$$

where F_0 is defined to be an image with all pixels as unity. However, since the default scheduling is dynamic, where a job is randomly distributed to some thread depending on the availability, this makes it difficult to implement (22). To overcome this problem, we have used static scheduling where the input image is exponentiated only n_T times if $K + 1$ is a multiple of n_T (once for each thread), or $n_T + 1$ times if $t n_T < K + 1 < (t + 1) n_T$. Here t is the largest integer less than or equal to $(K + 1)/n_T$. The subsequent images are computed by recursion (cf. lines 13 and 22 in Algorithm 4). We note that a thread can have private and shared variables. The private variables are stored in the reserved memory space of a thread. On the other hand, the shared variables are updated in the critical region to ensure that no two threads access the same variable.

4   Results

We now present some results obtained using the C implementation of Algorithm 4. In particular, we report few visual comparisons on natural images, and some statistics on the filtering accuracy and run-time of the implementation. We have used the grayscale versions of the standard images *House* and *Lena*, and the real-world image *Tiya* that was captured using a cellphone camera.

The visual comparisons are presented in Figures 5 and 6, where we have compared the output image from the fast algorithm with that of the bilateral filter. The two outputs are visually indistinguishable. For a quantitative comparison, we evaluated the root-mean-square error (RMSE) between the two images, which is given by

$$\text{RMSE} = \left\{ \frac{1}{|\Omega|} \sum_{\boldsymbol{i} \in \Omega} \left(\mathcal{B}[f](\boldsymbol{i}) - \tilde{\mathcal{B}}[f](\boldsymbol{i}) \right)^2 \right\}^{1/2}. \tag{23}$$

In Table 1, we have listed the RMSEs for the *Tiya* image for various values of σ_s and σ_r. The average RMSEs for four standard test images are listed in Table 2. In Figure 7, we present a visual comparison for coarse-to-fine values of the tolerance ϵ. Artifacts are clearly visible in the output image when ϵ is high, that is, when lesser number of terms are used in the Fourier approximation.

The run-times of the serial and parallel implementations are reported in Tables 3 and 4. The experiment was performed on an Intel quad-core 3.4 GHz machine with 32 GB memory. The reduction in computation time obtained using the parallel implementation is evident. Moreover, as expected,

---

Algorithm 4: Parallelized Fast Bilateral Filter.

Data: Image $f : \Omega \to \{0, 1, \ldots, M\}$;

Parameters: Filter parameters σ_s and σ_r, and tolerance ϵ;

Result: Output given by (11);

1 Set n_C as the number of physical cores of the system;

2 // *Number of threads*

3 $n_T \leftarrow n_C$;

4 **if** $n_C \geq 8$ **then**

5 $\quad \lfloor \; n_T \leftarrow 8$;

6 Map each thread to a distinct physical core of the system;

7 Compute T for the input image using the algorithm in [2];

8 $T_{\max} \leftarrow \max(T, 3.2\sigma_r)$;

9 $\nu \leftarrow 2\pi/(2T_{\max} + 1)$;

10 Based on ϵ, compute K and $\{\hat{g}(k) : k = 0, 1, \ldots, K\}$ using the rule in (10);

11 $t \leftarrow \lfloor (K+1)/n_T \rfloor$; // *Number of loops assigned to each thread*

12 // *Initializations*

13 $F_1 = \exp(-\iota\nu f)$;

14 $P \equiv 0$, $Q \equiv 0$;

15 // *Parallel implementation starts*

16 Create parallel regions for n_T threads, with variables k and t, and images P and Q shared;

17 Create a critical region in shared memory to execute lines 25 and 26;

18 **for** $k = 0, \ldots, K$ **do**

19 $\quad$ **if** $\mod (k, t) = 0$ **then**

20 $\quad \quad \lfloor \; F = \exp(-\iota k\nu f)$;

21 $\quad$ **else**

22 $\quad \quad \lfloor \; F = F \otimes F_1$;

23 $\quad P_k \leftarrow 2 \cdot \mathrm{real}(\hat{g}(k)\,(F \otimes \mathcal{G}[f \otimes F^*]))$;

24 $\quad Q_k \leftarrow 2 \cdot \mathrm{real}(\hat{g}(k)\,(F \otimes \mathcal{G}[F^*]))$;

25 $\quad P \leftarrow P \oplus P_k$;

26 $\quad Q \leftarrow Q \oplus Q_k$;

27 $\quad$ // *P_k and Q_k are deallocated*

28 // *Parallel implementation ends*

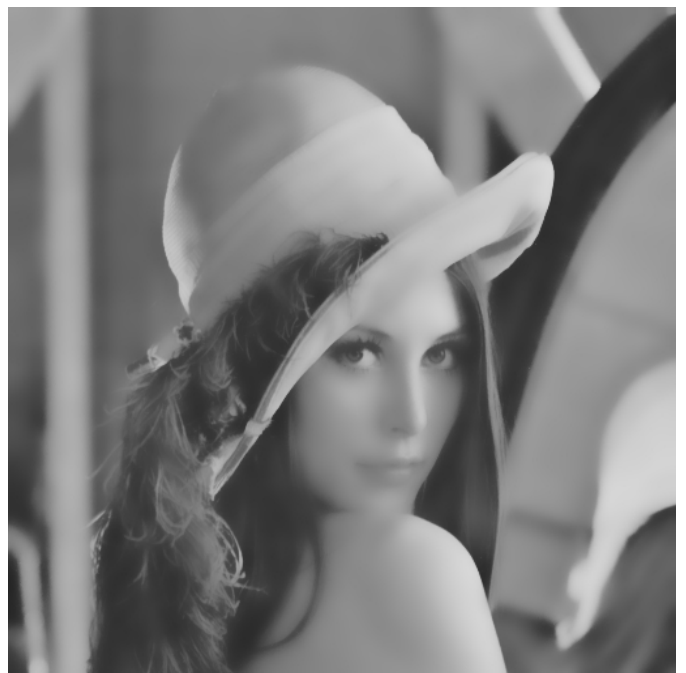**29** $\tilde{\mathcal{B}}[f] \leftarrow P \oslash Q$.

(a) Input.

(b) Exact.

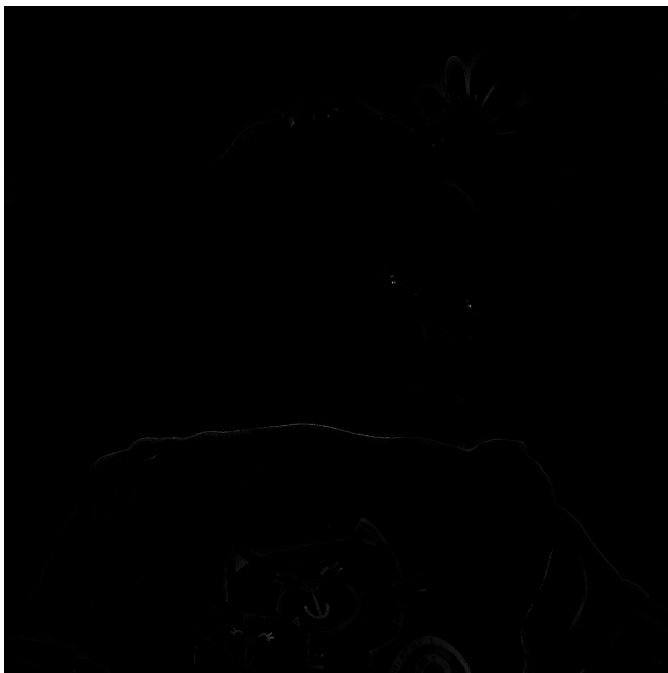(c) Error between (b) and (d).

(d) Fast.

Figure 5: Bilateral filtering of lena.png using $\sigma_s = 5, \sigma_r = 60$, and $\epsilon = 0.01$. The RMSE between (b) and (d) is 0.387291. The figure (c) shows the absolute value of the difference between (b) and (d). The range is mapped from $[0, 4]$ to $[0, 255]$. Notice that the visual artifacts in (c) are cluttered around sharp edges in the original image. This can be explained by the fact that the Fourier approximation is relatively poor (often assuming negative values) on the tails compared to that around the origin. Since the operating region for large pixel differences is precisely the tail, this can result in artifacts around edges. For example, see Figure 3, which shows ripples on the tails when $\epsilon = 0.01$.

(a) Input.

(b) Exact.

(c) Error between (b) and (d).

(d) Fast.

Figure 6: Bilateral filtering of tiya.jpg using $\sigma_s = 20, \sigma_r = 20$, and $\epsilon = 0.1$. The RMSE is 0.6168. The figure (c) shows the absolute value of the difference between (b) and (d). The range is mapped from $[0, 58]$ to $[0, 255]$. As in Figure 5, we see some visible artifacts around sharp edges.

(a) Direct.

(b) $\epsilon = 0.01$ ($K = 17$).

(c) $\epsilon = 1$ ($K = 0$).

(d) $\epsilon = 0.5$ ($K = 4$).

Figure 7: Comparison of different approximations of the bilateral filter for house.png using $\sigma_s = 3$ and $\sigma_r = 10$. When $\epsilon = 1$, the output image is visibly blurred since only the dc component is considered in this case. The result improves on reducing the tolerance to $\epsilon = 0.5$; however, the output still shows some artifacts since the kernel approximation is poor. When a sufficiently small tolerance of $\epsilon = 0.01$ is used, the proposed approximation becomes visually indistinguishable from the bilateral filter output.

σ_s \ σ_r	10	20	30	40	50	60	70	80	90	100
3	0.0598	0.0869	0.1072	0.1249	0.1412	0.1562	0.1689	0.0268	0.0283	0.0295
7	0.0638	0.1077	0.1439	0.1765	0.2050	0.2306	0.2517	0.0481	0.0510	0.0534
11	0.0572	0.0996	0.1367	0.1704	0.1988	0.2244	0.2456	0.0597	0.0634	0.0666

Table 1: RMSE for the *Tiya* image using $\epsilon = 0.001$.

σ_s \ σ_r	10	20	30	40	50	60	70	80	90	100
3	0.0816	0.1371	0.1849	0.2282	0.2714	0.1810	0.0482	0.0533	0.0584	0.0629
7	0.0768	0.1349	0.1925	0.2498	0.3080	0.3657	0.0704	0.0791	0.0862	0.0924
11	0.0730	0.1254	0.1776	0.2296	0.2817	0.3308	0.0837	0.0929	0.1005	0.1073

Table 2: Ensemble RMSE for the images *House*, *Cameraman*, *Lena*, and *Barbara* (see Image Credits) using $\epsilon = 0.001$.

σ_r	10	20	30	40	50	60	70	80	90	100
Serial	5891	3003	2190	1784	1376	1173	1172	1687	1687	1690
Parallel	2352	1409	1194	911	886	885	885	927	1280	905

Table 3: Run-times (in milliseconds) of the serial and parallel implementations for a megapixel image for fixed $\sigma_s = 3$ and $\epsilon = 0.001$.

σ_s	3	5	7	9	11	13	15	17	19	21
Time(ms)	948	934	953	968	978	987	997	1031	1042	1030

Table 4: Run-time of the parallel implementation (for a megapixel image) for different σ_s using the settings $\sigma_r = 40$ and $\epsilon = 0.001$. Notice that the speed essentially does not depend on σ_s.

the execution time does not scale with σ_s. Notice that we have not reported results in the regime $\sigma_r < 10$, since the bilateral filter virtually acts as an identity transformation in this case. On the other hand, the edge-preserving property of the filter is lost when $\sigma_r > 100$.

Acknowledgments

Image Credits

 Standard test images[6].

 CC-BY by Kunal N. Chaudhury.

---

[6] http://www.imageprocessingplace.com/root_files_V3/image_databases.htm

References

[1] K. N. Chaudhury, *Constant-time filtering using shiftable kernels*, IEEE Signal Processing Letters, 18 (2011), pp. 651–654. https://doi.org/10.1109/LSP.2011.2167967.

[2] ——, *Acceleration of the shiftable O(1) algorithm for bilateral filtering and nonlocal means*, IEEE Transactions on Image Processing, 22 (2013), pp. 1291–1300. https://doi.org/10.1109/TIP.2012.2222903.

[3] K. N. Chaudhury and S. D. Dabhade, *Fast and provably accurate bilateral filtering*, IEEE Transactions on Image Processing, 25 (2016), pp. 2519–2528. https://doi.org/10.1109/TIP.2016.2548363.

[4] K. N. Chaudhury, D. Sage, and M. Unser, *Fast O(1) bilateral filtering using trigonometric range kernels*, IEEE Transactions on Image Processing, 20 (2011), pp. 3376–3382. https://doi.org/10.1109/TIP.2011.2159234.

[5] R. Deriche, *Recursively implementating the Gaussian and its derivatives*, PhD thesis, INRIA, 1993.

[6] F. Durand and J. Dorsey, *Fast bilateral filtering for the display of high-dynamic-range images*, in ACM transactions on graphics (TOG), vol. 21, ACM, 2002, pp. 257–266.

[7] S. Ghosh and K. N. Chaudhury, *On fast bilateral filtering using Fourier kernels*, IEEE Signal Processing Letters, 23 (2016), pp. 570–573. https://doi.org/10.1109/LSP.2016.2539982.

[8] S. Paris and F. Durand, *A fast approximation of the bilateral filter using a signal processing approach*, European Conference on Computer Vision, (2006), pp. 568–580.

[9] S. Paris, P. Kornprobst, J. Tumblin, F. Durand, et al., *Bilateral filtering: Theory and applications*, Foundations and Trends® in Computer Graphics and Vision, 4 (2009), pp. 1–73.

[10] F. Porikli, *Constant time O(1) bilateral filtering*, in IEEE Conference on Computer Vision and Pattern Recognition, June 2008, pp. 1–8. https://doi.org/10.1109/CVPR.2008.4587843.

[11] K. Sugimoto and S. I. Kamata, *Compressive bilateral filtering*, IEEE Transactions on Image Processing, 24 (2015), pp. 3357–3369. https://doi.org/10.1109/TIP.2015.2442916.

[12] C. Tomasi and R. Manduchi, *Bilateral filtering for gray and color images*, in IEEE International Conference on Computer Vision, IEEE, 1998, pp. 839–846.

[13] Q. Yang, K. H. Tan, and N. Ahuja, *Real-time O(1) bilateral filtering*, in IEEE Conference on Computer Vision and Pattern Recognition, June 2009, pp. 557–564. https://doi.org/10.1109/CVPR.2009.5206542.

[14] I.T. Young and L.J. Van Vliet, *Recursive implementation of the Gaussian filter*, Signal processing, 44 (1995), pp. 139–151.