Fast L^1 Gauss 2D Image Transforms

Dina Bashkirova[1,2*], Shin Yoshizawa[1§], Rustam Latypov[2†], Hideo Yokota[1¶]

[1]Image Processing Research Team
RIKEN Center for Advanced Photonics, RIKEN
2-1, Hirosawa, Wako, Saitama, 351-0198, Japan
Email: {*dina.bashkirova,§shin,¶hyokota}@riken.jp

[2]Institute of Computational Mathematics and Information Technologies
Kazan Federal University
420008, Kazan, 35 Kremlyovskaya
Email: †roustam.latypov@kpfu.ru

***Abstract*—Gaussian convolution has many science and engineering applications, and is widely applied to computer vision and image processing tasks. Due to its computational expense and rapid spreading of high quality data (bit depth/dynamic range), accurate approximation has become important in practice compared with conventional fast methods. In this paper we propose a novel approximation method for fast Gaussian convolution of 2D images. Our method employs L^1 distance metric to achieve fast computations while preserving high accuracy. Our numerical experiments show the advantages over conventional methods in terms of speed and precision.**

***Keywords*—Gaussian Smoothing, Laplace Distribution, Fast Approximation Algorithms.**

I. Introduction

Gaussian convolution is a core tool in mathematics and many related research areas, such as probability theory, physics, and signal processing. Gauss transform is a discrete analogue to the Gaussian convolution, and has been widely used for many applications including kernel density estimation [1] and image filtering [2]. Despite its reliable performance and solid theoretical foundations, Gauss transform in its exact form along with other kernel-based methods has a drawback – it is very computationally expensive (has quadratic computational complexity w.r.t. the number of points) and hard to scale to higher dimensions. Which is why there have been many attempts to overcome these problems by creating approximation algorithms, such as fast Gauss transform [3], dual-tree fast Gauss transforms [4], fast KDE [5], and Gaussian kd-trees [6]. Also, box kernel averaging [7] and recursive filtering [8] have been popular in computer graphics and image processing because of their simplicity, see the surveys [9], [10] for numerical comparisons of these approximation methods.

Since high bit depth (also dynamic range) images have become popular in both digital entertainment and scientific/engineering applications, it is very important to acquire high approximation precision and to reduce artifacts cased by drastic truncation employed in many conventional methods focused on computational speed. One of the highly accurate methods is called fast L^1 Gauss transform approximation [11] based on using L^1 distance instead of conventional L^2 Euclidean metric. This L^1 metric preserves most of the properties of the L^2 Gaussian, and is separable, hence it allows to perform computations along each dimension separately, which is very beneficial in terms of computational complexity. Also, L^1 Gaussian has only one peak in Fourier domain at the coordinate origin, and therefore its convolution does not have some undesirable artifacts that box kernels and truncation methods usually have. However, this algorithm works only on one-dimensional (1D) point sets, although it can be extended to uniformly distributed points in higher dimensions by performing it separately in each dimension. In order to be able to acquire Gauss transform for non-uniformly distributed two-dimensional points and to further generalize it to higher dimensional cases, we need to extend existing method [11] to the 2D uniform case.

In this paper we propose a novel approximation method for fast Gauss two-dimensional (2D) image transform. Our method is based on extending the fast L^1 Gauss transform approximation on uniformly distributed 2D points that allows to perform Gaussian convolution quickly while preserving high accuracy. We demonstrate that efficiency of the proposed method in terms of computational complexity, numerical timing, and approximation precision.

II. Fast L^1 Gauss Transform

In this section, we briefly describe the 1D domain splitting algorithm [11] employed for fast L^1 Gauss transforms.

Consider the ordered point set $\mathbb{X} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}$, $x_i \geq x_{i-1}$, $\forall i = \overline{2, N}$. Each point x_i has a corresponding value $I_i \in \mathbb{R}$, e.g. pixel intensity in case of images. The L^1 Gauss transform for each point in set $\mathbb{X}$ is given by

$$J(x_j) = \sum_{i=1}^{N} G(x_j - x_i) I_i, \quad G(x) = \exp(-\frac{|x|}{\sigma}), \quad (1)$$

where $G(x)$, $x \in \mathbb{R}$, is a L^1 Gaussian function (also called Laplace distribution in statistics) with its standard deviation σ. It is convenient to decompose L^1 norm by splitting its domain by using the point x_1 such that

$$|x_j - x_i| = \begin{cases} |x_j - x_1| - |x_i - x_1| & \text{if } x_1 \leq x_i \leq x_j, \\ |x_i - x_1| - |x_j - x_1| & \text{if } x_1 \leq x_j \leq x_i. \end{cases} \quad (2)$$

Thus, Gauss transform (1) using the equation (2) becomes

$$\begin{aligned} J(x_j) = I_i + G(x_j - x_1) \sum_{i=1}^{j-1} \frac{I_i}{G(x_i - x_1)} + \\ + \frac{1}{G(x_j - x_1)} \sum_{i=j+1}^{N} G(x_i - x_1) I_i. \end{aligned} \quad (3)$$

Such representation (3) allows to reduce the amount of computational operations, since values $G(x_j - x_1)$, $\frac{1}{G(x_j - x_1)}$, and the sums $\sum_{i=1}^{j-1} \frac{I_i}{G(x_i - x_1)}$ and $\sum_{j+1}^{N} I_i G(x_i - x_1)$ can be precomputed in linear time. However, using the equation (3) may imply some numerical issues, such as overflow, if the distance between x_1 and x_l, $l \in \{i, j\}$ is relatively large. To avoid such issues, this algorithm introduced certain representative points (poles) $\{\alpha_k \in \mathbb{R}\}$ instead of using the single point x_1, where the distance between α_k and x_l is smaller than the length that causes the numerical instability. Hence the equation (3) becomes more complex form, a highly accurate truncation can be applied where $G(\alpha_k - x_j)$ is equal to numerically zero, see [11] for further technical details.

Although this algorithm can be used in case of multidimensional images by applying it separately in each dimension, this separable implementation approach is not applicable to non-uniformly distributed high-dimensional point sets. Therefore we present a novel and natural extension of the domain splitting concept on 2D cases (images) in the following sections.

III. Two-dimensional Algorithm

For a given 2D point set $\mathbb{X} = \{\mathbf{x}_i\}_{i=1}^N$, $\mathbf{x}_i = (x_i, y_i) \in \mathbb{R}^2$, L^1 distance between two points in $\mathbb{R}^2$ is given by $|\mathbf{x}_j - \mathbf{x}_i| = |x_j - x_i| + |y_j - y_i|$, thus the Gauss transform (1) is represented by the formula:

$$J(\mathbf{x}_j) = \sum_{i=1}^{N} \exp(-\frac{|x_j - x_i| + |y_j - y_i|}{\sigma}) I_i.$$

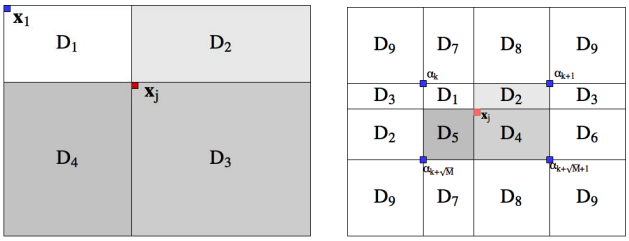
Domain splitting (2) for 2D points is given by

$$|x_j - x_i| + |y_j - y_i| =$$

$$\begin{cases} |x_j - x_1| - |x_i - x_1| + |y_j - y_1| - |y_i - y_1| & \text{if } \mathbf{x}_i \in D_1 \\ |x_i - x_1| - |x_j - x_1| + |y_j - y_1| - |y_i - y_1| & \text{if } \mathbf{x}_i \in D_2 \\ |x_j - x_1| - |x_i - x_1| + |y_i - y_1| - |y_j - y_1| & \text{if } \mathbf{x}_i \in D_3 \\ |x_i - x_1| - |x_j - x_1| + |y_i - y_1| - |y_j - y_1| & \text{if } \mathbf{x}_i \in D_4, \end{cases}$$

$$\begin{aligned} D_1 &= \{\mathbf{x}_i | x_1 \le x_i \le x_j, y_1 \le y_i \le y_j\}, \\ D_2 &= \{\mathbf{x}_i | x_1 \le x_j \le x_i, y_1 \le y_i \le y_j\}, \\ D_3 &= \{\mathbf{x}_i | x_1 \le x_i \le x_j, y_1 \le y_j \le y_i\}, \\ D_4 &= \{\mathbf{x}_i | x_1 \le x_j \le x_i, y_1 \le y_j \le y_i\}, \end{aligned}$$

see Fig. 1a for geometric illustration of the domains.



(a) Single pole $\mathbf{x}_1$ case (b) Multipole $\{\alpha_k\}$ case

Fig. 1: Illustration of 2D domain splliting.

Using the above decomposition, Gauss transform is represented similar to (3):

$$\begin{aligned} J(\mathbf{x}_j) = I(\mathbf{x}_j) + F(x_j)F(y_j) \sum_{\mathbf{x}_i \in D_1(j)} \frac{1}{F(x_i)F(y_i)} I(\mathbf{x}_i) + \\ + \frac{F(x_j)}{F(y_j)} \sum_{\mathbf{x}_i \in D_2(j)} \frac{F(y_i)}{F(x_i)} I(\mathbf{x}_i) + \frac{F(y_j)}{F(x_j)} \sum_{\mathbf{x}_i \in D_3(j)} \frac{F(x_i)}{F(y_i)} I(\mathbf{x}_i) + \\ + \frac{1}{F(x_j)F(y_j)} \sum_{\mathbf{x}_i \in D_4(j)} F(x_i)F(y_i) I(\mathbf{x}_i), \end{aligned} \tag{4}$$

where $F(x_j) \equiv G(x_j - x_1)$ and $F(y_j) \equiv G(y_j - y_1)$.

Precomputation and storage of values $F(x_j)F(y_j)$, $\frac{F(x_j)}{F(y_j)}$, $\frac{1}{F(x_j)F(y_j)}$, and $\frac{F(y_j)}{F(x_j)}$ require $O(4N)$ operations and $O(4N)$ space, and all the subsequent sums can be iteratively computed in $O(N)$ operations. Gauss transform for all points using the formula (4) requires $O(10N)$ as opposed to employing the separable implementation of equation (3) for $O(6N)$ operations. Since computing the Gauss transform using the equation (4) is numerically troublesome, it is reasonable to divide the space into smaller groups and perform computations separately, as it was proposed in [11]. Let us introduce a novel 2D multipole approach for solving this problem.

Consider a set of poles $\{\alpha_k\}_{k=1}^M$, $\alpha_k = (a_k, b_k) \in \mathbb{R}^2$. The distance between points in $\mathbb{X}$ using poles α_k is given by $|\mathbf{x}_i - \mathbf{x}_j| =$

$$\begin{cases} |x_i - a_k| - |x_j - a_k| + |y_i - b_k| - |y_j - b_k| & \text{if } \mathbf{x}_i \in D_1 \\ |x_j - a_k| - |x_i - a_k| + |y_i - b_k| - |y_j - b_k| & \text{if } \mathbf{x}_i \in D_2 \\ |x_i - a_k| + |x_j - a_k| + |y_i - b_k| - |y_j - b_k| & \text{if } \mathbf{x}_i \in D_3 \\ |x_i - a_k| - |x_j - a_k| + |y_j - b_k| - |y_i - b_k| & \text{if } \mathbf{x}_i \in D_4 \\ |x_j - a_k| - |x_i - a_k| + |y_j - b_k| - |y_i - b_k| & \text{if } \mathbf{x}_i \in D_5 \\ |x_i - a_k| + |x_j - a_k| + |y_j - b_k| - |y_i - b_k| & \text{if } \mathbf{x}_i \in D_6 \\ |x_i - a_k| - |x_j - a_k| + |y_i - b_k| + |y_j - b_k| & \text{if } \mathbf{x}_i \in D_7 \\ |x_j - a_k| - |x_i - a_k| + |y_i - b_k| + |y_j - b_k| & \text{if } \mathbf{x}_i \in D_8 \\ |x_i - a_k| + |x_j - a_k| + |y_i - b_k| + |y_j - b_k| & \text{if } \mathbf{x}_i \in D_9, \end{cases}$$

where

$$D_1 = \{\mathbf{x}_i | x_i \in D_1^x, y_i \in D_1^y\}, D_2 = \{\mathbf{x}_i | x_i \in D_2^x, y_i \in D_1^y\},$$

$$D_3 = \{\mathbf{x}_i | x_i \in D_3^x, y_i \in D_1^y\}, D_4 = \{\mathbf{x}_i | x_i \in D_1^x, y_i \in D_2^y\},$$

$$D_5 = \{\mathbf{x}_i | x_i \in D_2^x, y_i \in D_2^y\}, D_6 = \{\mathbf{x}_i | x_i \in D_3^x, y_i \in D_2^y\},$$

$$D_7 = \{\mathbf{x}_i | x_i \in D_1^x, y_i \in D_3^y\}, D_8 = \{\mathbf{x}_i | x_i \in D_2^x, y_i \in D_3^y\},$$

$$D_9 = \{\mathbf{x}_i | x_i \in D_3^x, y_i \in D_3^y\},$$

$$\begin{aligned} D_1^x &= \{x_i | a_k \le x_i \le x_j \text{ or } x_j \le x_i \le a_k\}, \\ D_2^x &= \{x_i | a_k \le x_j \le x_i \text{ or } x_i \le x_j \le a_k\}, \\ D_3^x &= \{x_i | x_i \le a_k \le x_j \text{ or } x_j \le a_k \le x_i\}, \\ D_1^y &= \{y_i | b_k \le y_i \le y_j \text{ or } y_j \le y_i \le b_k\}, \\ D_2^y &= \{y_i | b_k \le y_j \le y_i \text{ or } y_i \le y_j \le b_k\}, \\ D_3^y &= \{y_i | y_i \le b_k \le y_j \text{ or } y_j \le b_k \le y_i\}, \end{aligned}$$

$$J(\mathbf{x}_j) = I_j + \mathcal{G}(x_j)\mathcal{G}(y_j) \sum_{\mathbf{x}_i \in D_1} \frac{I_i}{\mathcal{G}(x_i)\mathcal{G}(y_i)} + \frac{1}{\mathcal{G}(x_j)\mathcal{G}(y_j)} \sum_{\mathbf{x}_i \in D_5} \mathcal{G}(x_i)\mathcal{G}(y_i) I_i + \frac{\mathcal{G}(y_j)}{\mathcal{G}(x_j)} \sum_{\mathbf{x}_i \in D_2} \frac{\mathcal{G}(x_i)}{\mathcal{G}(y_i)} I_i + \frac{\mathcal{G}(x_j)}{\mathcal{G}(y_j)} \sum_{\mathbf{x}_i \in D_4} \frac{\mathcal{G}(y_i)}{\mathcal{G}(x_i)} I_i +$$
$$+ \sum_{k \in D_9} A_k^j + \sum_{k \in D_7} B_k^j + \sum_{k \in D_8} C_k^j + \sum_{k \in D_3} D_k^j + \sum_{k \in D_6} E_k^j, \tag{5}$$

$$A_k^j = \mathcal{G}(x_j)\mathcal{G}(y_j) \sum_{\mathbf{x}_i = (k)}^{(k+1)^{-1}} \mathcal{G}(x_i)\mathcal{G}(y_i) I_i, \qquad B_k^j = \mathcal{G}(x_j)\mathcal{G}(y_j) \sum_{\mathbf{x}_i = (k)}^{(k+1)^{-1}} \frac{\mathcal{G}(y_i)}{\mathcal{G}(x_i)} I_i, \qquad C_k^j = \frac{\mathcal{G}(y_j)}{\mathcal{G}(x_j)} \sum_{\mathbf{x}_i = (k)}^{(k+1)^{-1}} \mathcal{G}(x_i)\mathcal{G}(y_i) I_i,$$

$$D_k^j = \mathcal{G}(x_j)\mathcal{G}(y_j) \sum_{\mathbf{x}_i = (k)}^{(k+1)^{-1}} \frac{\mathcal{G}(x_i)}{\mathcal{G}(y_i)} I_i, \qquad E_k^j = \frac{\mathcal{G}(x_j)}{\mathcal{G}(y_j)} \sum_{\mathbf{x}_i = (k)}^{(k+1)^{-1}} \mathcal{G}(x_i)\mathcal{G}(y_i) I_i.$$

see Fig. 1b for geometric illustration of the domains with their poles. The point $\mathbf{x}_j$ is assigned for one representative pole defined by

$$\alpha_k(\mathbf{x}_j) = \max_k \{\alpha_k | a_k \le x_j, b_k \le y_j\},$$

which is the closest pole to $\mathbf{x}_j$ that has absolute values of coordinate smaller than $\mathbf{x}_j$.

For each point $\mathbf{x}_j$, the multipole L^1 Gauss transform is given by the equation (5) where $\mathcal{G}(x_j) \equiv G(x_j - a_k)$, $\mathcal{G}(y_j) \equiv G(y_j - b_k)$, and $\lambda(\cdot)$ is an index function defined by

$$\lambda(k) = \min_{1 \le j \le N} (\mathbf{x}_j | a_k \le x_j < a_{k+1} \text{ and } b_k \le y_j < b_{k+1}).$$

For the sake of simplicity, we assume that the numbers of poles in 2D are same M. Following [11], M and the poles $\{\alpha_k\}$ are given by

$$\{a_k\} = \{b_k\} = \frac{\{0, 1, 2, ..., (M-1)\}w}{M}, \tag{6}$$
$$w = \max(|x_1 - x_N|, |y_1 - y_N|), \; M = [\frac{w}{\varphi\sigma \log(\text{MAX})}],$$

where $[\cdot]$ is the ceiling function, MAX is the maximum value of precision (e.g., double floating point: DBL_MAX in C programming language), and φ is a user-specified parameter (0.5 is employed in our numerical experiments). The above pole selection scheme leads to $\max(G(a_{k+1} - a_k), G(b_{k+1} - b_k)) <$ MAX which theoritically guarantees numerical stability in our method.

If the distance between poles is determined by the equation (6) and $G(\alpha_k - \mathbf{x}_j)$ becomes numerically zero if $|\alpha_k - \mathbf{x}_j| > \frac{w}{\varphi M}$, we can efficiently truncate Gauss transform by approximating the values:

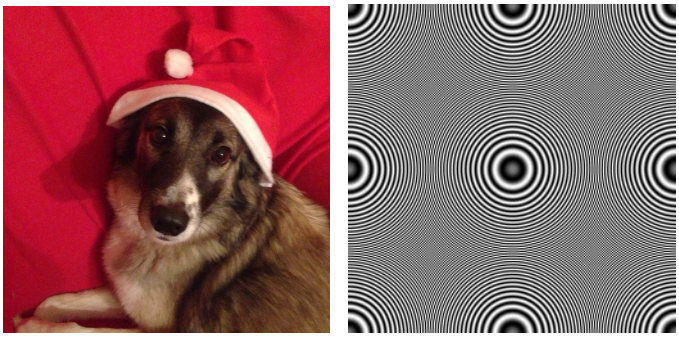
$$\sum_{\alpha_k \in D_9} A_k^j \approx \sum_{\alpha_k \in \mu(D_9)} A_k^j, \sum_{\alpha_k \in D_7} B_k^j \approx \sum_{\alpha_k \in \mu(D_7)} B_k^j,$$
$$\sum_{\alpha_k \in D_8} C_k^j \approx \sum_{\alpha_k \in \mu(D_8)} C_k^j, \sum_{\alpha_k \in D_3} D_k^j \approx \sum_{\alpha_k \in \mu(D_3)} D_k^j,$$
$$\sum_{\alpha_k \in D_6} E_k^j \approx \sum_{\alpha_k \in \mu(D_6)} E_k^j,$$

where $\mu(D_*) = \{\mathbf{x}_i \in D_* \mid |\alpha_k(\mathbf{x}_j) - \alpha_k(\mathbf{x}_i)| \le \frac{w}{\varphi M}\}$. In other words, instead of computing terms $A_k^j, B_k^j, C_k^j, D_k^j, E_k^j$ across all the corresponding point sets, we take into account only the neighbouring points, which allows to avoid nested loop structure in our implementation and speed up the computational process.

As in the 1D algorithm [11], the terms can be iteratively computed in linear time. Assume that an image consists of $\sqrt{N} \times \sqrt{N}$ pixels and the number of poles along each dimension is equal to M, total complexity of our method is equal to $O(16N + 2\frac{\sqrt{N}}{M} + 4\frac{N}{M^2})$, which is a little bit slower than the separable implementation employed in [11] that requires $O(12N + 2\sqrt{N} + M)$ operations.

IV. Numerical Experiments

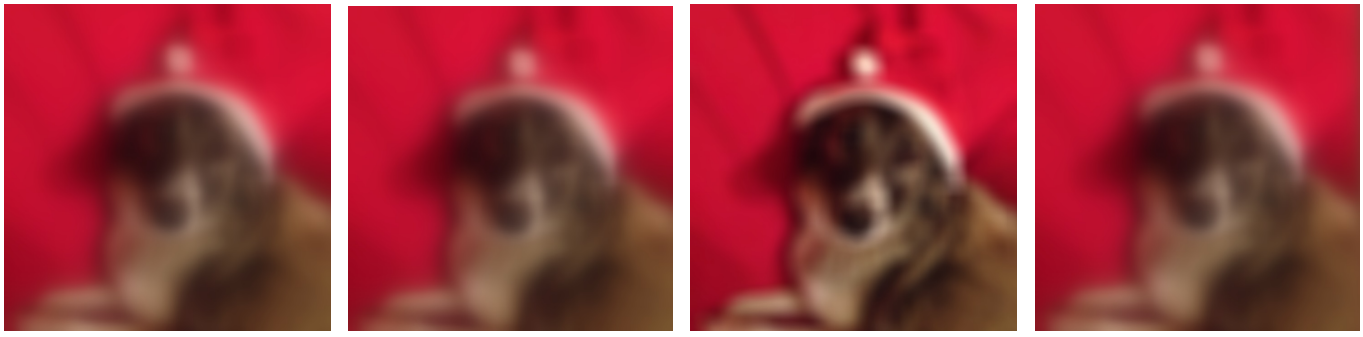
We held all the experiments on Intel Core i7-6600U 2.60 GHz dual core computer with 16GB RAM and a 64-bit operating system. We compared the multipole version of our algorithm with box kernel (Box) using moving average method [7], the 1D domain splitting (YY14) with separable implementations [11], and Fast Discrete Cosine Transform (FDCT) via the FFT package [12] well-known for its efficiency.



(a) Input image 1 (b) Input image 2

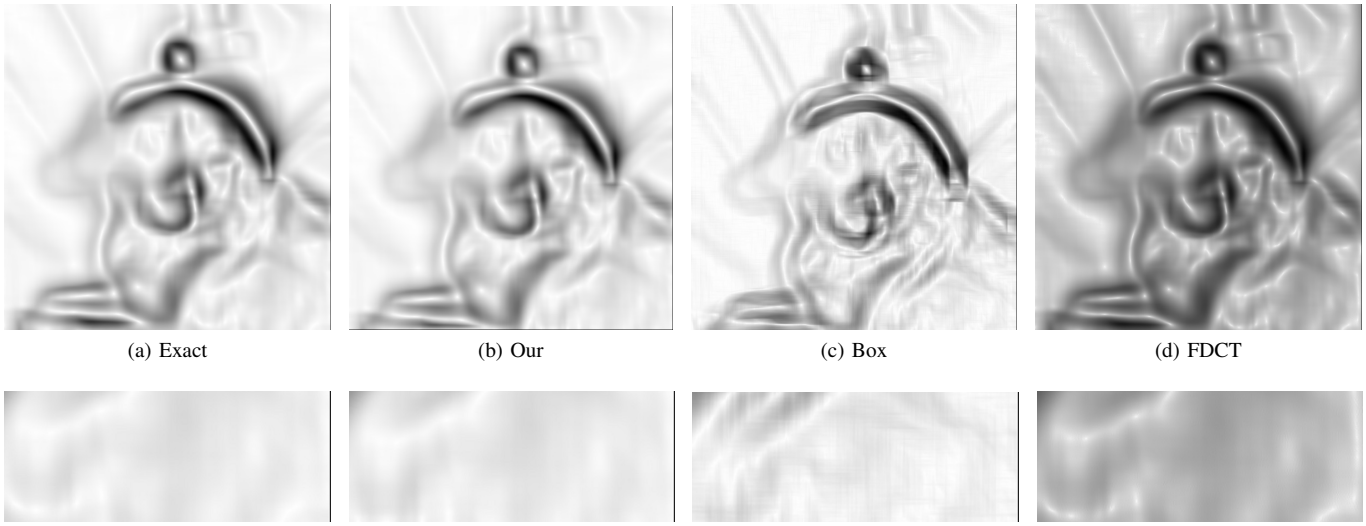
Fig. 2: Input images.

To evaluate the performance of the methods mentioned above we used randomly generated 2D point sets with 10 different sizes from 128^2 to 5120^2 and 10 various values of $\sigma = 5, 10, ..., 50$. The radius for the Box method was chosen equal to σ. The timing results (see Fig. 5) show that our method is slightly slower than the 1D domain splitting (YY14) despite its theoretical complexity is much larger. It is worth noticing



(a) Exact (b) Our (c) Box (d) FDCT

Fig. 3: Results of smoothing ($\sigma = 20$).



(a) Exact (b) Our (c) Box (d) FDCT

(e) Exact (f) Our (g) Box (h) FDCT

Fig. 4: Visualisation of $|\nabla I|$ for comparison of artifacts ($\sigma = 20$).

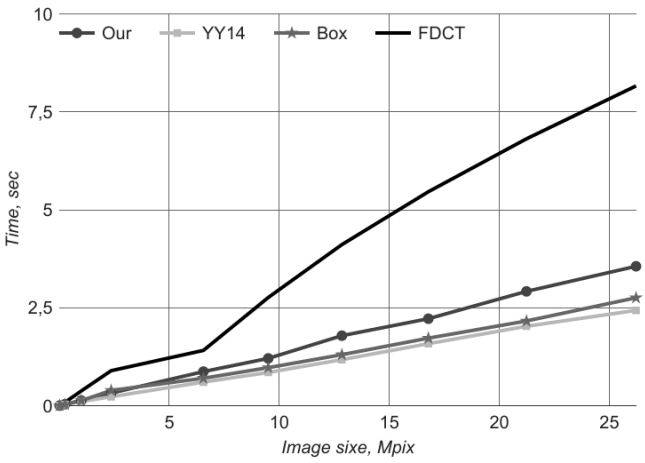


Fig. 5: Timing with respect to image size (averaged by σ).

that the implementation of our method can be further improved by using GPU-based or parallel computing techniques.

However, accuracy evaluation results (see Table I) show that our method achieves best approximation quality among the discussed methods. We evaluate the precision using $E_{\max}$ and PSNR measures. Consider I^e is the exact L^1 Gauss transform result, I^a is the approximation achieved by a given algorithm, and $d_i = |I_i^e - I_i^a|$, $E_{\max}$ is calculated using formula $E_{\max} = \max\limits_{1 \le i \le N} d_i$. We also use peak signal-to-noise ratio (PSNR) [2] to measure the performance of our algorithm according to the equation

$$\text{PSNR} = -10 \log(\sum_{i=1}^{N} (\frac{d_i}{\max(I_i^e, I_i^a)})^2).$$

We performed linear image smoothing by the following normalized convolutions for each color channel:

$$\frac{\int G(\mathsf{x} - \mathsf{y}) I(\mathsf{y}) d\mathsf{y}}{\int G(\mathsf{x} - \mathsf{y}) d\mathsf{y}} \to \frac{J(\mathsf{x}_j)}{\sum_i^N G(\mathsf{x}_j - \mathsf{x}_i)}$$