IPOL IMAGE PROCESSING ON LINE

Published in Image Processing On Line on 2014–11–17.
Submitted on 2013–03–13, accepted on 2014–04–29.
ISSN 2105–1232 © 2014 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
http://dx.doi.org/10.5201/ipol.2014.79

The Heeger-Bergen Pyramid-Based Texture Synthesis Algorithm

Thibaud Briand[1], Jonathan Vacher[2], Bruno Galerne[3], Julien Rabin[4]

[1] Département de mathématiques, ENS Cachan, France (tbriand@ens-cachan.fr)
[2] CEREMADE, CNRS-Université Paris-Dauphine, France (jonathan.vacher@ceremade.dauphine.fr)
[3] Laboratoire MAP5 (UMR CNRS 8145), Université Paris Descartes, France (bruno.galerne@parisdescartes.fr)
[4] GREYC, CNRS-ENSICAEN, Université de Caen, France (julien.rabin@unicaen.fr)

Abstract

This contribution deals with the Heeger-Bergen *pyramid-based texture analysis/synthesis* algorithm. It brings a detailed explanation of the original algorithm tested on many characteristic examples. Our analysis reproduces the original results, but also brings a minor improvement concerning non-periodic textures. Inspired by visual perception theories, Heeger and Bergen proposed to characterize a texture by its first-order statistics of both its color and its responses to multiscale and multi-orientation filters, namely the *steerable pyramid*. The Heeger-Bergen algorithm consists in the following procedure: starting from a white noise image, histogram matchings are performed to the image alternately in the image domain and the steerable pyramid domain, so that the corresponding output histograms match the ones of the input texture.

Source Code

An on-line demo[1] of the Heeger-Bergen pyramid-based texture synthesis algorithm is available. The demo permits to upload a color image to extract a subimage and to run the texture synthesis algorithm on this subimage.

The algorithm available in the demo is a slightly improved version treating non-periodic textures by a "periodic+smooth" decomposition [13]. The algorithm works with color textures and is able to synthesize textures with larger size than the input image. The original version of the Heeger-Bergen algorithm (where the boundaries are handled by mirror symmetrization) is optional in the source code.

An ANSI C implementation is available for download here[2]. It is provided with:

- An illustrated html documentation;
- Source code;

This code requires `libpng`, `libfftw3`, `openmp`, and `getopt`. Compilation and usage instructions are included in the `README.txt` file of the zip archive.

The illustrated HTML documentation can be reproduced from the source code by using doxygen (see the `README.txt` file of the zip archive for details).

Keywords: texture; synthesis; heeger; bergen

[1] http://dx.doi.org/10.5201/ipol.2014.79
[2] http://dx.doi.org/10.5201/ipol.2014.79

1 Introduction

Given an input texture image, the aim of texture synthesis algorithms is to produce an output texture image that is both visually similar to and pixel-wise different from the input texture. Texture synthesis algorithms can be separated in two categories: neighborhood-based methods and statistical constraint approaches.

Neighborhood-based algorithms consist in producing a new texture by arranging local neighborhoods of the input texture in a consistent way. The first methods of this kind create the new texture one pixel at a time [4, 22] while subsequent algorithms would rather copy a whole neighborhood or patch [3, 9]. We refer to the state of the art [23] for a more complete survey of this category of texture synthesis algorithms.

Contrary to neighborhood-based algorithms, synthesis algorithms based on statistical constraints attempt at modeling the texture by involving statistical and/or perceptual considerations. They typically consist in two steps, the analysis step and the synthesis step. The analysis step estimates a set of statistics from the input texture (e.g. histograms, covariance function, etc.). The synthesis step generates a random image that satisfies the statistical constraints estimated during the analysis step. The Heeger-Bergen algorithm belongs to this category of algorithms. It proposes to characterize a texture by the first-order statistics of both its color and its responses to multiscale and multi-orientation filters, namely the *steerable pyramid*, which is motivated by texture discrimination theories (see [8] and the references therein). The Heeger-Bergen approach has been extended and improved by several authors [15, 14, 17] by taking into account second-order or higher statistics and/or by using more involved multiscale image representations. A simpler approach taken in [6] models certain textures as Gaussian random fields. Tartavel et al. recently proposed an hybrid neighborhood-based algorithm relying on a spectral statistical constraint [20].

Another field related to texture synthesis is procedural texture synthesis by example. Procedural textures are programs that permit to define and generate a texture in a continuous domain They are used in computer graphics in situations where one does not want to deal with raster images (made of pixels) [10]. Within this field, a simplified version of the Heeger-Bergen algorithm adapted to the procedural texture framework has been proposed [11]. More recently, a similar algorithm relying on Gaussian random fields models has been developed [7].

Let us now describe in more detail the Heeger-Bergen texture synthesis algorithm. The output texture is initialized with a white noise image. Histogram matchings are performed to this output texture image alternately in the image domain and a multiscale transform domain, namely the *steerable pyramid*, until all the output histograms match the ones of the input texture.

The paper is organized as follows. Section 2 is dedicated to the full presentation of the algorithm. We first present the two fundamental tools of the Heeger-Bergen algorithm, the steerable pyramid decomposition and histogram matching. We then describe in detail the algorithm for grayscale and color textures. Finally we discuss the artifacts caused by the non periodicity of the input texture. We propose a new alternative to deal with this issue by replacing the input texture by its periodic component [13], which enables us to attenuate a low frequency artifact inherent to the Heeger-Bergen algorithm.

Section 3 presents several experiments whose goal is to illustrate the influence of the algorithm parameters.

2 Algorithm

2.1 Steerable Pyramid

2.1.1 General Description and Illustration

The *steerable pyramid* is a linear multiscale and multi-orientation image decomposition that has been developed in the 90s by E. Simoncelli and his co-authors (see e.g. [19, 18] and the references therein). Given an input image, it is obtained by first splitting the image into a high frequency part and a low frequency part and then by sequentially applying bandpass oriented filters to the low frequency image followed by downsampling. This results in a sequence of images having different sizes, referred to as a pyramid, each corresponding to a certain scale and orientation, apart from the high frequency and the low frequency residuals. Two examples of steerable pyramid decomposition are displayed in figures 1 and 2.

In this paper we use the (real version of the) filters described by Portilla and Simoncelli [15]. This slightly differs from the original paper [8] where the real steerable pyramid was computed with some approximation, while with the filters used here, the transformation is made in the Fourier domain without making use of any approximation. Still, as in the original paper, we only consider the real part of the steerable pyramid. Indeed, one can only deal with real filter responses to apply histogram matching in the synthesis step of the algorithm. In what follows, we describe precisely how these filters are defined and implemented.

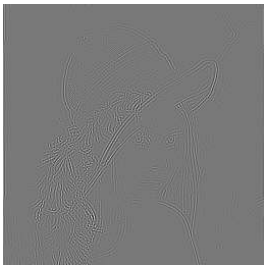
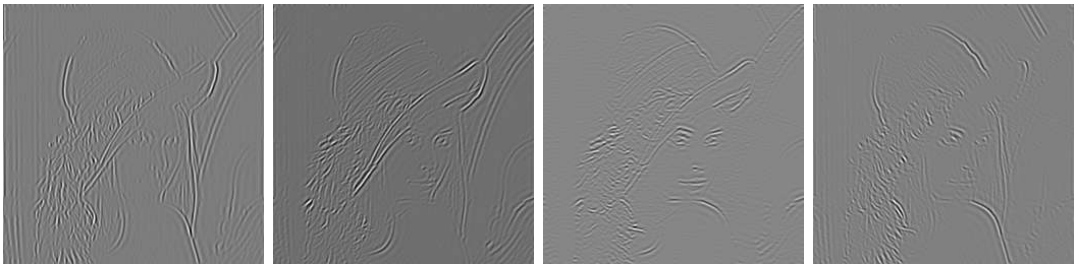
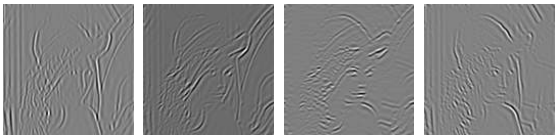
Original image	Associated steerable pyramid
	 High frequency residual  1st scale of oriented subbands with angle $\theta = 0$, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$  2nd scale of oriented subbands with angle $\theta = 0$, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$  Low frequency residual

Figure 1: Steerable pyramid decomposition of the Lena image with two scales and four orientations.

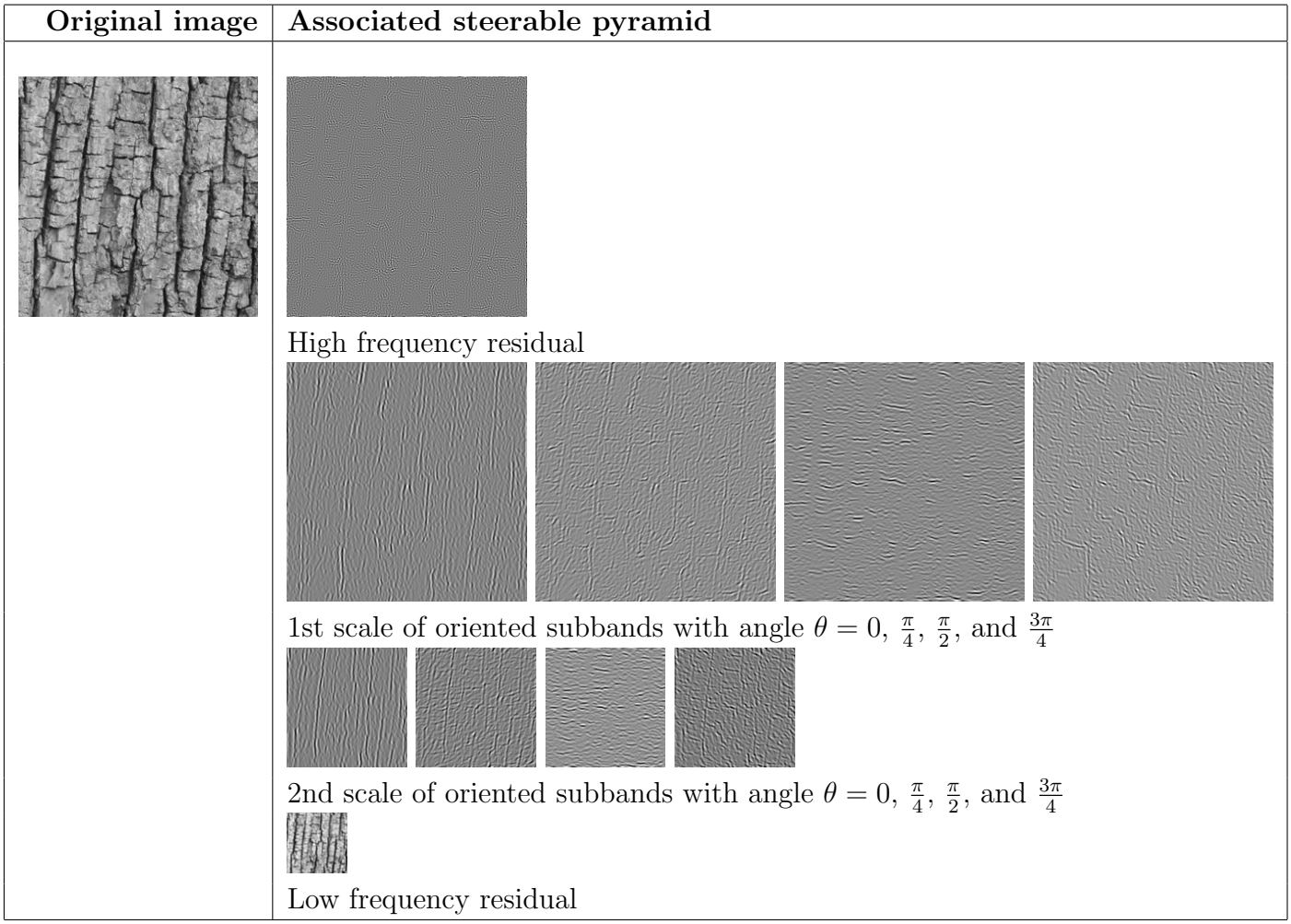


Figure 2: Steerable pyramid decomposition of a texture image with two scales and four orientations

2.1.2 Multiscale Decomposition Procedure

The steerable pyramid decomposition convolves recursively the image with oriented filters at different scales. The decomposition has two parameters:

- the number of scales P (default value 4),
- the number of orientations Q (default value 4).

The steerable pyramid decomposition algorithm consists of a sequence of convolution products with several filters. The involved filters h_0, l_0, l and b_q will be explicitly defined in the next section, for now let us just mention that h_0 is a high-pass filter, l_0, l are low-pass filters, and that each b_q is an oriented bandpass filter. The decomposition algorithm is detailed in Algorithm 1. It is also represented by the diagram of Figure 3. As clarified by Algorithm 1 and illustrated by figures 1 and 2, the steerable pyramid of an image u of size $M \times N$ is made of $PQ + 2$ images:

- A high frequency residual of size $M \times N$.
- For each scale $p = 1, \ldots, P$, Q images of size $M/2^{p-1} \times N/2^{p-1}$.
- A low frequency residual of size $M/2^P \times N/2^P$.

Algorithm 1: Steerable pyramid decomposition

Input : Number of scales P, number of orientations Q, discrete image u of size $M \times N$ such that M and N are multiples of 2^P.

Output: Steerable pyramid of u: sequence of $PQ + 2$ images of varying size (see figures 1 and 2 for illustration)

1. Compute the high frequency residual $h_0 * u$ and store it in the pyramid.
2. Compute the low frequency band $v \leftarrow l_0 * u$.
3. **for** scale $p = 1$ **to** P **do**
4. Compute the oriented high frequency band $b_q * v$ for each orientation $q = 0, \dots, Q-1$ and store it in the pyramid (these Q images constitute the p-th scale of the pyramid).
5. $v \leftarrow$ the low frequency image $l * v$.
6. Downsample v by a factor of two.
7. **end**
8. Store the remaining image v (of size $M/2^P \times N/2^P$) as the low frequency residual of the pyramid.

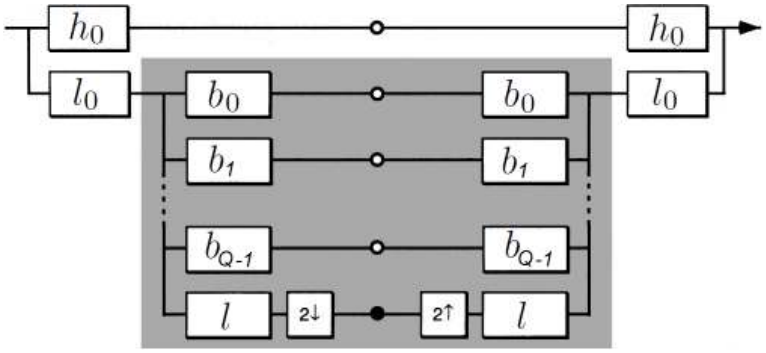


Figure 3: Diagram for the steerable pyramid: The left part corresponds to the steerable pyramid image decomposition. The right part corresponds to the image recontruction from the pyramid. This recursive step is performed until the number of desired pyramid scales P is reached.

Note that for the pyramid decomposition to be well-defined, it is necessary that the width M and the height N of the input image u be multiples of 2^P. This will be assumed throughout, even though it should be mentioned that this restriction could be overcome by adapting the downsampling and upsampling procedure for images with odd sizes. However our current implementation does not enable the pyramid decomposition if the width and the height are not multiples of 2^P. Thus, a preliminary crop is performed on the sample texture to ensure this condition.

All the convolution products of the pyramid decomposition algorithm are performed using the fast Fourier transform (FFT) for computational efficiency. The downsampling operation is also performed in the Fourier domain by extracting the centered part of the Fourier transform (which corresponds exactly to downsampling since it is always applied to band-limited images).

2.1.3 Definition and Computation of the Steerable Filters

In this section we define explicitly the filters involved in the steerable pyramid computation. We recall that in this paper we use the real version of the complex filters described by Portilla and

Simoncelli [15][3].

As said above, when computing the steerable pyramid, all the convolution products are performed by component-wise multiplications in the Fourier domain. Hence, one only needs an expression of the discrete Fourier transforms (DFT) of the filter images h_0, l_0, l, b_q. Not surprisingly, these filters are actually directly defined in the Fourier domain by sampling various functions defined analytically in the continuous frequency domain.

For a discrete image of size $M \times N$, we denote by $\Omega_{M,N} = \{0, \dots, M-1\} \times \{0, \dots, N-1\}$ the discrete image domain (that is, the pixel indices). We denote by $\hat{\Omega}_{M,N} = \left\{-\frac{M}{2}, \dots, \frac{M}{2}-1\right\} \times \left\{\frac{N}{2}, \dots, \frac{N}{2}-1\right\}$ the discrete Fourier domain (that is the frequency indices). The continuous frequency domain will be $[-\pi, \pi]^2$. The following definition of the Discrete Fourier Transform (DFT) is used

$$\left(\mathcal{F}_{M,N}(u)\right)_{m,n} = \hat{u}_{m,n} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} u_{k,l} e^{-2i\pi\left(k\frac{m}{M} + l\frac{n}{N}\right)}, \quad (m,n) \in \hat{\Omega}_{M,N},$$

and its associated inverse DFT is

$$\left(\mathcal{F}^{-1}_{M,N}(v)\right)_{k,l} = \frac{1}{MN} \sum_{m=-\frac{M}{2}}^{\frac{M}{2}-1} \sum_{n=-\frac{N}{2}}^{\frac{N}{2}-1} v_{m,n} e^{2\pi i\left(m\frac{k}{M} + n\frac{l}{N}\right)}, \quad (k,l) \in \Omega_{M,N}.$$

Finally, since the filters will be defined in terms of polar coordinates in the continuous frequency domain $[-\pi, \pi]^2$, let us define the standard polar transformation ρ to go from Cartesian coordinates to polar ones[4]

$$\begin{array}{rccl} \rho: & [-\pi,\pi]^2 & \longrightarrow & \mathbb{R}\times]-\pi,\pi] \\ & (x,y) & \longmapsto & (r,\theta) = \begin{cases} (|x|, \pi) & \text{if } y = 0 \text{ and } x \leq 0, \\ \left(\sqrt{x^2+y^2}, 2\arctan\left(\frac{y}{x+\sqrt{x^2+y^2}}\right)\right) & \text{otherwise.} \end{cases} \end{array}$$

The functions defined in the continuous frequency domain $[-\pi, \pi]^2$ are at the heart of the definition of the steerable pyramid filters. For any polar coordinate $(r, \theta) \in \mathbb{R}\times]-\pi, \pi]$ define

$$L(r,\theta) = L(r) = \begin{cases} 1 & \text{if } r \leqslant \frac{\pi}{4}, \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{4r}{\pi}\right)\right) & \text{if } \frac{\pi}{4} \leqslant r \leqslant \frac{\pi}{2}, \\ 0 & \text{if } r \geqslant \frac{\pi}{2}, \end{cases}$$

and

$$H(r,\theta) = H(r) = \begin{cases} 0 & \text{if } r \leqslant \frac{\pi}{4}, \\ \cos\left(\frac{\pi}{2}\log_2\left(\frac{2r}{\pi}\right)\right) & \text{if } \frac{\pi}{4} \leqslant r \leqslant \frac{\pi}{2}, \\ 1 & \text{if } r \geqslant \frac{\pi}{2}, \end{cases}$$

which correspond respectively to a low pass filter and a high pass filter. For the Q orientation indices $q = 0, \dots, Q-1$, define also the following cone-shaped filters

$$G_q(r,\theta) = G_q(\theta) = \alpha_Q \left(\cos\left(\theta - \frac{\pi q}{Q}\right)^{Q-1} \mathbb{1}_{\left|\theta - \frac{\pi q}{Q}\right| \leqslant \frac{\pi}{2}} + \cos\left(\theta - \frac{\pi(q-Q)}{Q}\right)^{Q-1} \mathbb{1}_{\left|\theta - \frac{\pi(q-Q)}{Q}\right| \leqslant \frac{\pi}{2}} \right),$$

[3]The formula given in [15] for the definition of the polar function L contains a small error of factor of two. The corrected formulas reproduced here can be found in Appendix A of a later article [16].

[4]In practice this transformation is straightforward thanks to the C function `atan2`.

where the normalizing constant α_Q is given by

$$\alpha_Q = 2^{Q-1}\frac{(Q-1)!}{\sqrt{Q(2(Q-1))!}}.$$

Then, the high, low and oriented filters are defined by

$$\begin{aligned} L_0(r,\theta) &= L_0(r) = L\left(\frac{r}{2}\right), \\ H_0(r,\theta) &= H_0(r) = H\left(\frac{r}{2}\right), \\ B_q(r,\theta) &= H(r)G_q(\theta). \end{aligned}$$

Given an image u, to compute the convolution $u*v$ using FFT one only needs the expression of the DFT of v. The DFTs of the filter images h_0, l_0, l, b_q, which we will denote respectively by $\hat{h}_0, \hat{l}_0, \hat{l}, \hat{b}_q$, are defined as follows: For each filter $f \in \{h_0, l_0, l, b_q\}$, the DFT $\hat{f}$ is obtained by sampling the corresponding analytical function $F \in \{H_0, L_0, L, B_q\}$ in the continuous frequency domain according to the formula

$$\hat{f}_{m,n} = F \circ \rho\left(\frac{2\pi m}{M}, \frac{2\pi n}{N}\right), \quad (m,n) \in \hat{\Omega}_{M,N}. \tag{1}$$

The function $F \circ \rho$ is sampled M times along the x-axis and N-times along the y-axis, as illustrated by Figure 4.

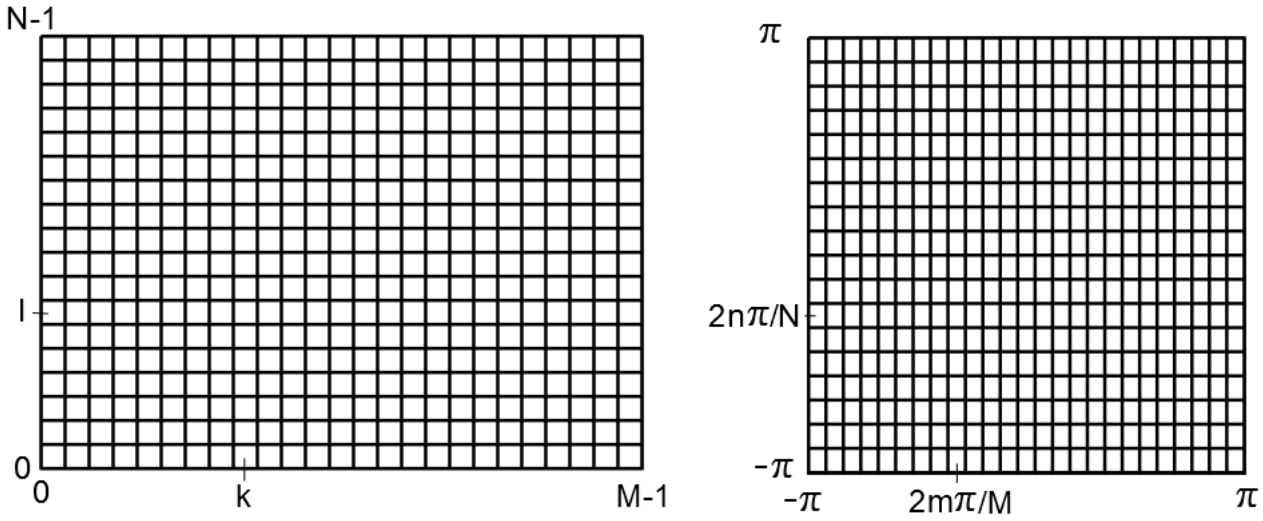


Figure 4: Discrete spatial domain $\Omega_{M,N}$ and corresponding sampling grid of the continuous frequency domain $[-\pi,\pi]^2$ used for the definition of the DFT of the discrete pyramid filters. When M is different from N the Fourier sampling grid cells are rectangular.

In Algorithm 1, each convolution of the form $f*v$ (or $f*u$) where f is one of the filters $\{h_0, l_0, l, b_q\}$ is performed in the Fourier domain as follows:

1. Compute the DFT $\left(\hat{f}_{m,n}\right)$, $(m,n) \in \hat{\Omega}_{M,N}$, by sampling F according to Equation (1) where $M \times N$ is the size of the second image v.

2. Component-wise multiply $\hat{f}$ and $\hat{v}$ (which gives the DFT of $f * v$).

The down-sampling is also performed in the Fourier domain by extracting the central part of the DFT (which is zero outside this region before down-sampling). In the end, the number of FFT calls used by the pyramid decomposition algorithm (Algorithm 1) is $PQ + 3$:

- One forward FFT to compute $\hat{u}$.
- $PQ + 2$ backward FFTs (involving images of different sizes) to compute the pyramid images.

2.1.4 Steerable Pyramid Reconstruction

One of the interesting features of the steerable pyramid is that it has been designed to be self-inverting [18]. This means that reconstructing any image from its pyramid is possible and only involves the very same filters as for the decomposition computation.

The steerable pyramid reconstruction procedure corresponds to the right part of the diagram of Figure 3. The reconstruction algorithm is given in detail in Algorithm 2. As for Algorithm 1, we describe the operations in terms of convolution in the image domain, but in practice all the operations are performed as component-wise multiplications in the Fourier domain. In particular the upsampling operation must be performed in the Fourier domain by extending the DFT with zeros in the high frequencies, in order to be consistent with the decomposition procedure. In the spatial domain, this corresponds to a zero-padding zoom by a factor of two. The number of FFT calls for the reconstruction algorithm is the same as for the decomposition algorithm, namely $PQ + 3$.

Algorithm 2: Steerable pyramid reconstruction

Input : A steerable pyramid having P scales, Q orientations, and corresponding image size $M \times N$ (see Figures 1 and 2 for illustration)

Output: An image u of size $M \times N$

1. $u \leftarrow$ low frequency residual.
2. **for** scale $p = P$ **to** 1 **do**
3. Upsample u by a factor of two using zero-padding zoom (that is, extending the DFT of u with zeros in the high frequencies)
4. $u \leftarrow u * l$.
5. For each orientation $q = 0, \ldots, Q-1$, convolve the subband at scale p and orientation q with its corresponding filter b_q and add the obtained image to u.
6. **end**
7. $u \leftarrow u * l_0$.
8. Convolve the high frequency residual with h_0 and add the obtained image to u.
9. Return u.

The steerable filters are designed so that each stage of the diagram of Figure 3 has a flat system response [18]. More precisely here we have for all $(r, \theta) \in \mathbb{R}\times]-\pi, \pi]$,

$$H_0(r,\theta)^2 + L_0(r,\theta)^2 = 1$$

and for all $(r, \theta) \in [0, \pi]\times]-\pi, \pi]$,

$$\sum_{k=0}^{Q-1} B_q(r,\theta)^2 + L(r,\theta)^2 = 1$$

(here putting the functions to the square corresponds to the double convolutions that occur in the spatial domain when applying the decomposition followed by the reconstruction, see Figure 3). This

ensures that if the input of the reconstruction algorithm is the steerable pyramid of an image u, then the output of the reconstruction algorithm is exactly the image u.

In terms of linear operators, if one denotes by A the matrix of the pyramid decomposition operator, one easily sees that the matrix of the reconstruction operator is simply the transposed A^T of the decomposition matrix A. However, this reconstruction matrix A^T also coincides with the pseudo-inverse $A^\dagger$ of the decomposition matrix A. Indeed, since $A^T A = Id$, the pseudo-inverse of A is $A^\dagger = \left(A^T A\right)^{-1} A^T = A^T$. In short, the pyramid reconstruction operator is the pseudo-inverse of the decomposition operator [18]. This observation is of importance here since the reconstruction operator will be applied to pyramids that do not belong to the range of A (i.e. to pyramids that do not correspond exactly to the decomposition of one image). Indeed, after applying histogram matching to each image of a pyramid, there is no guarantee that the resulting pyramid still lies in the range of A. Since the reconstruction algorithm corresponds to the pseudo-inverse of the decomposition, one can argue that it is the natural reverse operation even when applied to pyramids that are outside the range of A.

2.2 Histogram Matching

Given an input image u and a reference image v of the same size, histogram matching consists in changing the gray-level values of the input u so that it gets the same histogram as the reference image v.

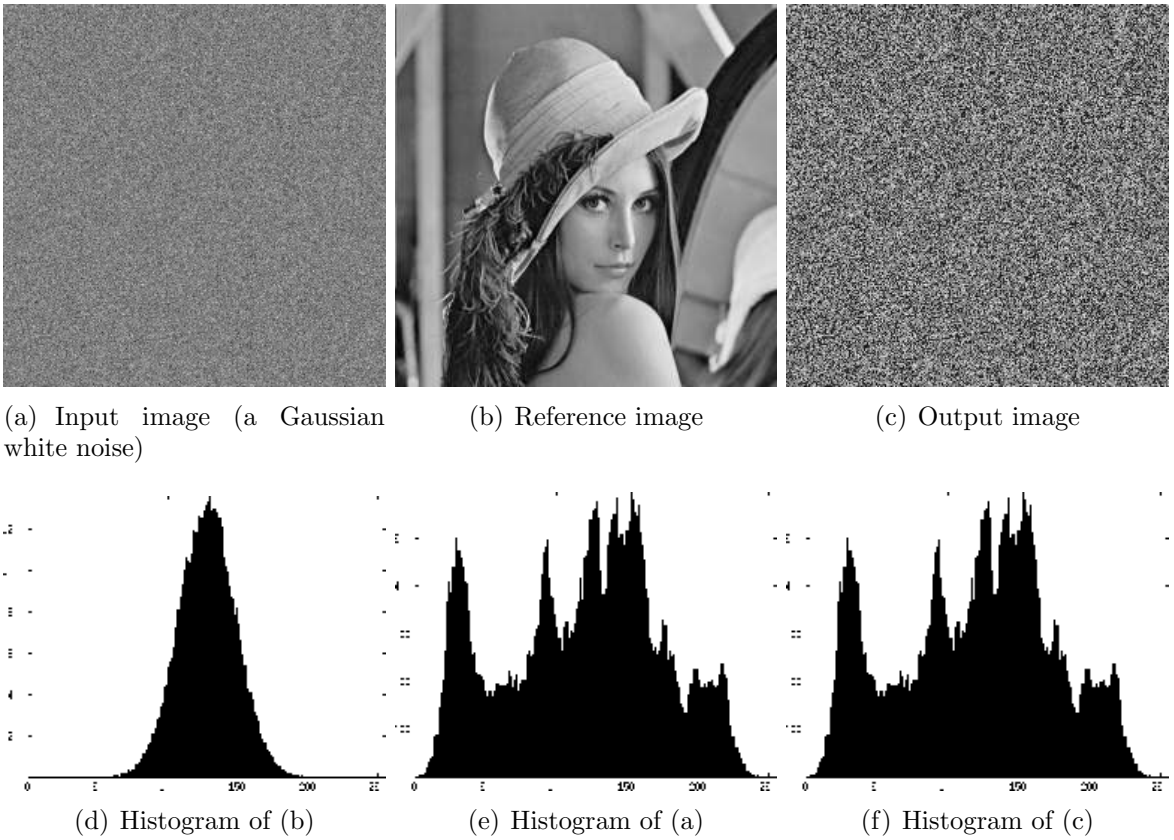
Histogram matching is known to be an ill-posed problem [2] since several solutions coexist. Our histogram matching algorithm sorts the pixel list of the two images u and v, and assigns to the pixel of u having rank k the gray-value of the pixel of v having rank k. This simple algorithm gives a "perfect histogram matching", in the sense that the output has exactly the same histogram as the reference image while the relative rank of the input pixel is preserved. Besides, it is theoretically well-founded since it corresponds to the optimal transport plan between the two discrete measures $\sum_{m,n} \delta_{u_{m,n}}$ and $\sum_{m,n} \delta_{v_{m,n}}$ [21, Chapter 1, Page 25].

The complete histogram matching procedure is summarized in Algorithm 3. Examples of histogram matchings are shown in Figures 5 and 6. If the reference image v is used several times for histogram matching, one should store its sorting permutation τ to save computation time (which is done in our implementation). The histogram matching procedure proposed here is different from the one described in the Heeger-Bergen paper [8], which is subject to several unnecessary quantizations.

Algorithm 3: Histogram matching

Input : Input image u, reference image v (both images have size $M \times N$)
Output: Image u having the same histogram as v (the input u is lost)

1. Define $L = MN$ and describe the images as vectors of length L (e.g. by reading values line by line).
2. **Sort the reference image v:**
3. Determine the permutation τ such that $v_{\tau(1)} \leq v_{\tau(2)} \leq \cdots \leq v_{\tau(L)}$.
4. **Sort the input image u:**
5. Determine the permutation σ such that $u_{\sigma(1)} \leq u_{\sigma(2)} \leq \cdots \leq u_{\sigma(L)}$.
6. **Match the histogram of u:**
7. **for** rank $k = 1$ **to** L **do**
8. $\quad u_{\sigma(k)} \leftarrow v_{\tau(k)}$ (the k-th pixel of u takes the gray-value of the k-th pixel of v).
9. **end**



(a) Input image (a Gaussian white noise) (b) Reference image (c) Output image

(d) Histogram of (b) (e) Histogram of (a) (f) Histogram of (c)

Figure 5: Example of histogram matching: The histogram of a Gaussian white noise is matched with the histogram of the Lena image.

More generally, one can also match histograms of images having different sizes if the input image length and width are multiples of the ones of the reference image. In this case, if for example the reference v has size $M \times N$ and the input image has size $rM \times rN$, with $r = 2, 3, \ldots$, the matching is done by calling Algorithm 3 with the input u and the larger reference image w obtained by replicating r^2 times each pixel of v. Equivalently, and more efficiently, this amounts to replacing line 8 of Algorithm 3 by

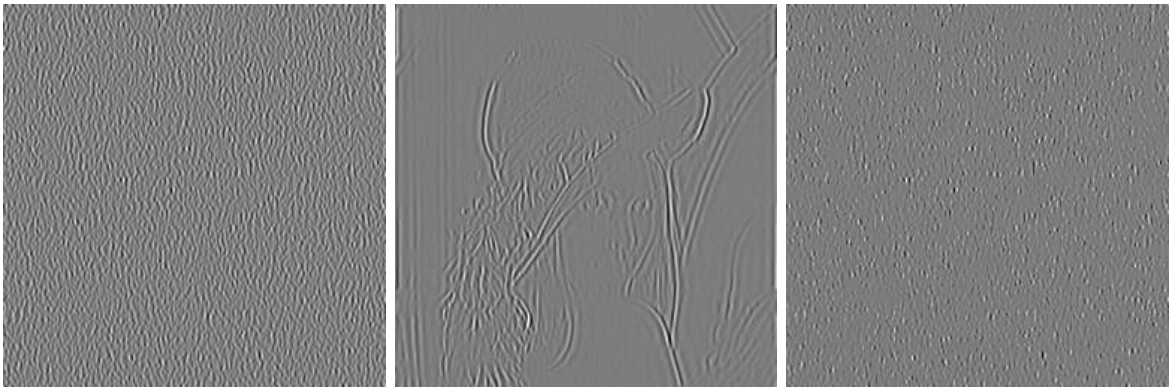
$$u_{\sigma(r^2(k-1)+i)} \leftarrow v_{\tau(k)}, \quad \text{for } i = 1, 2, \ldots, r^2,$$

that is, the k-th block of r^2 pixels of u takes the gray-value of the k-th pixel of v.

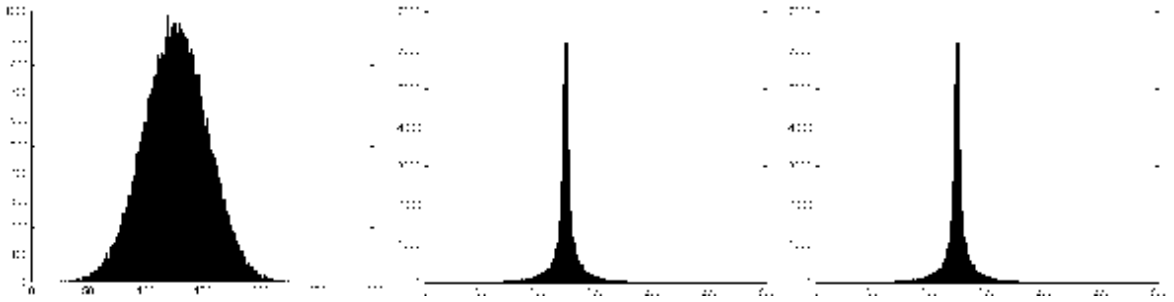
One known drawback of performing histogram matching with this method is that, for pixels that have the same gray-level in the input image u, the final gray-level assignment depends on the scanning order of the sorting procedure. However, since in practice we use this procedure with input images u that takes float values (starting with a white noise image), the probability that several pixels of u have the same gray-level is negligible. Hence we neglect the effect of this artifact, even though practical solutions exist[5].

A classical alternative for histogram matching is to apply the pseudo-inverse of the cumulative distribution functions of v to the distribution function of u. One drawback of this method is that it requires a convention for the pseudo-inverse of the cumulative distribution function, and it does not

[5]To avoid this artifact, one can randomly permute the pixels having the same gray-level. Alternatively, one can add a small noise to the gray-values of u before sorting, similarly as in the method of [1]. The result of this random dequantization is similar to using a random scanning order along groups of pixels having similar gray-level value (and thus imposing total order on the pixels [2]), but is much easier to implement.



(a) Input image (a pyramid image of a Gaussian white noise) (b) Reference image (a pyramid image of Figure 1) (c) Output image



(d) Histogram of (b) (e) Histogram of (a) (f) Histogram of (c)

Figure 6: Example of histogram matching of a pyramid image: The histogram of an oriented pyramid image of a Gaussian white noise is matched with the corresponding image in Lena's pyramid (see Figure 1).

necessarily lead to perfect histogram matching. However, one advantage is that it can be applied to images having different sizes (without the multiplicity constraint described above).

2.3 Texture Synthesis Algorithm for Grayscale Textures

We can now present the texture synthesis algorithm developed by Heeger and Bergen [8]. Starting from a white noise image, histogram matchings are performed to the texture image alternatively in the image domain and in the steerable pyramid domain. The motivation is that after a few iterations, all the output histograms will match the ones of the input texture, and thus according to several psychophysical experiments (see [8] and the references therein), the output texture will be visually similar to the input texture. The algorithm is detailed in Algorithm 4.

An additional extension of the algorithm is to synthesize an output texture that is larger than the input texture, under the condition that the output image length and width are multiples of the input image length and width. In this situation, the steps of the algorithm are unchanged, except that the histogram matching steps involve images having different sizes (as explained in the previous section).

2.4 Extension to Color Images

The Heeger-Bergen algorithm relies on histogram matching and thus it is only well-defined for grayscale images.

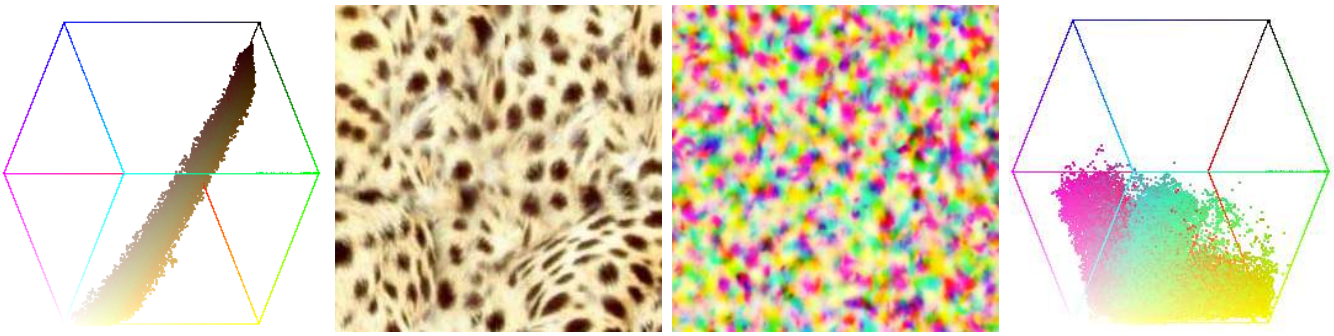
Algorithm 4: Heeger-Bergen texture synthesis algorithm for grayscale images (without extension)

Input : Number of scales P, number of orientations Q, texture image u of size $M \times N$ such that M and N are multiples of 2^P, number of iterations N_{iter}

Output: Texture image v of size $M \times N$

1. **Input analysis:**
2. Compute and store the steerable pyramid with P scales and Q orientations of the input texture u.
3. **Output synthesis:**
4. Initialize v with a Gaussian white noise.
5. Match the gray-level histogram of v with the gray-level histogram of the input u.
6. **for** iteration $i = 1$ **to** N_{iter} **do**
7. Compute the steerable pyramid of v.
8. For each of the $PQ + 2$ images of this pyramid, apply histogram matching with the corresponding image of the pyramid of u.
9. Apply the image reconstruction algorithm to this new histogram-matched pyramid and store the obtained image in v.
10. Match the gray-level histogram of v with the gray-level histogram of the input u.
11. **end**
12. Return v.

A first naive idea to extend the algorithm to RGB color textures is to apply it on each color channel of the input texture. However synthesizing a color texture by considering that its three RGB color channels are independent is not satisfying. Indeed, RGB color channels are highly correlated, and consequently synthesizing a color texture by synthesizing independently its three color channels does not respect the channels correlations, which leads to the creation of wrong colors. This is illustrated by Figure 7 where the Heeger-Bergen algorithm for grayscale textures (see Algorithm 4) is applied independently to the three channels of an input color texture.



(a) Color cube of (b) (b) Input texture (c) Output texture (d) Color cube of (c)

Figure 7: Independent channel synthesis in RGB space: The Heeger-Bergen algorithm for grayscale textures (see Algorithm 4) is applied independently to the three RGB channels of the input color texture (b). Notice the presence of wrong colors as attested by the differences between the RGB point clouds of the two textures (the filtering process of the on-line demo [12] has been used to display the RGB point clouds).

The solution proposed by Heeger and Bergen [8] is to change the RGB color space into an input-

adapted color space in which the independent channel synthesis gives good results. This new color space is obtained by a principal component analysis (PCA) of the RGB point cloud corresponding to the input texture. Intuitively, in the PCA color space, the color channels are decorrelated[6] and thus nearly independent. Hence making the channels independent in the output texture does not affect much the output quality.

We know describe in detail how the PCA color space is computed. For an RGB color image $u = (u_R, u_G, u_B)$ of size $M \times N$ it consists in the following operations:

- Compute the mean value $m = (m_R, m_B, m_G)$ of each RGB color channel.
- List the RGB pixel values of the centered image $u - m$ in a matrix $A \in \mathcal{M}_{3,MN}(\mathbb{R})$.
- Compute the correlation matrix $C = \frac{1}{MN-1} AA^t$.
- Diagonalize this 3×3 real symmetric matrix: set $C = PDP^T$ where D is the diagonal matrix containing the non-negative eigenvalues **in decreasing order** and P is the matrix of the corresponding orthogonal change of basis matrix.

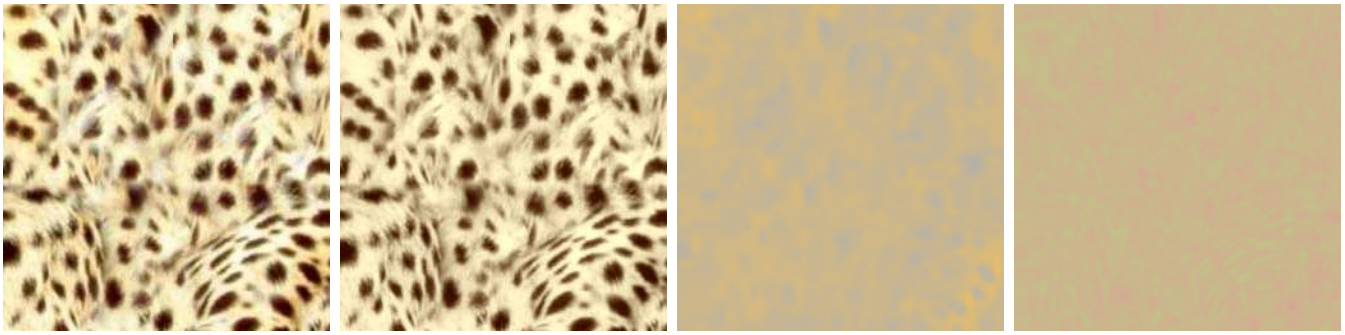
The complexity of this operation is linear in the number of pixels. Once the change of basis matrix P is computed, to go from one RGB image $v = (v_R, v_G, v_B)$ to its representation in the PCA color space of u, one does:

- change each RGB pixel $(v_{R,i}, v_{G,i}, v_{B,i})^T$ into $P^T(v_{R,i} - m_R, v_{G,i} - m_G, v_{B,i} - m_B)^T$.

Obviously, the reverse operation to go from an image $v = (v_1, v_2, v_3)$ given in the PCA color space to the corresponding image in RGB color space is

- change each PCA pixel $(v_{1,i}, v_{2,i}, v_{3,i})^T$ into $(m_R, m_B, m_G)^T + P(v_{1,i}, v_{2,i}, v_{3,i})^T$.

A PCA channel decomposition of a color texture is shown in Figure 8. Observe that the dynamic of the texture is mainly contained in the first principal component.



(a) Color texture (b) 1st principal component (c) 2nd principal component (d) 3rd principal component

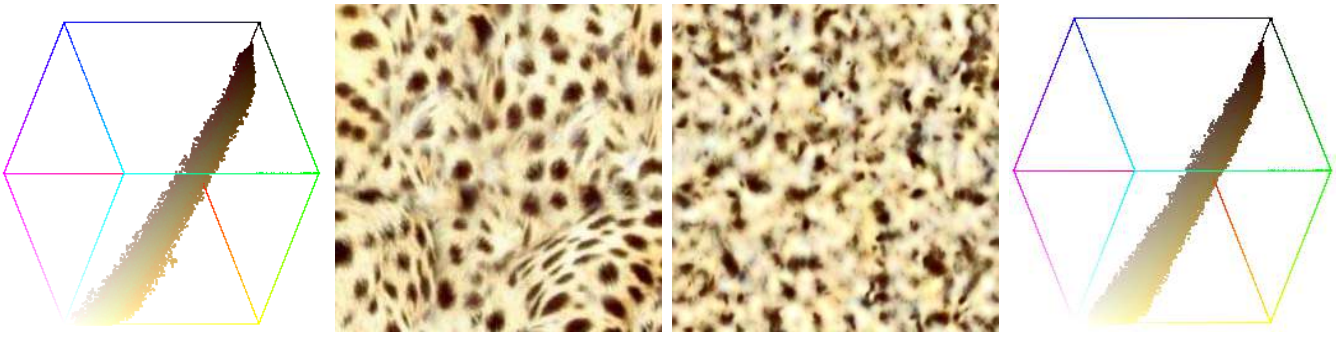
Figure 8: PCA channels of a color texture (for visualization, the mean color of the texture is added to each component). Note that the dynamic of the texture is mainly contained in the first principal component. This observation is valid for most standard texture samples.

The Heeger-Bergen texture synthesis algorithm for RGB color textures is the following:

[6]Let us mention that the PCA channels are not totally decorrelated, only their intensity values are. This is because the PCA only considers color correlations and not spatial correlations. See [7, Section 7] for a more detailed discussion and the definition of an alternative color space for independent channel synthesis.

1. Compute the PCA color space of the input image u.
2. Determine the channels of u in the PCA color space.
3. Apply the texture synthesis Algorithm 4 on each PCA channel. This gives an output texture v in the PCA color space.
4. Convert the image v in the RGB color space by applying the procedure described above. The obtained RGB image is the output of the algorithm.

An example of synthesis of an RGB texture using this algorithm is shown in Figure 9. One can observe that thanks to the use of the PCA color space, the output texture does not present noticeable wrong colors.



(a) Color cube of (b). (b) Texture sample. (c) Texture synthesized. (d) Color cube of (c).

Figure 9: Independent channel synthesis in PCA space: The Heeger-Bergen algorithm for grayscale textures (see Algorithm 4) is applied independently to the three PCA channels of the input color texture (b). Observe that the colors of the output texture are similar to the ones of the input, as confirmed by the similarity between the two corresponding RGB color cubes.

2.5 Avoiding Artifacts Due to Non Periodicity

Since the pyramid decomposition computation is based on the DFT, it treats the input image as if periodic. However, in practice real-world textures are never periodic. Consequently this periodic edge handling artificially introduces discontinuities in the input texture. This may create artifacts in the output texture as illustrated by the second image of Figure 11.

To cope with this problem, Heeger and Bergen propose to use a mirror symmetrization at the border. Although this solution ensures that the input texture is indeed continuous at the border, it is not perfect since it artificially introduces new orientations in the input texture (one should think of the example of a slanted wood texture such as the one of Figure 11).

Following a previous approach [6], we propose instead to replace the input texture by its *periodic component*, as defined by Moisan's "periodic + smooth" image decomposition [13]. We recall in Appendix A how the periodic component of an image is computed. The "periodic + smooth" image decomposition is illustrated by Figure 10. In most cases, the periodic component is close to the original texture, except for some color gradation. Indeed, the gradations are suppressed since they lead to edge discontinuities.

We illustrate with Figure 11 the results of the texture synthesis algorithm with three different edge handling: No edge handling, mirror symmetrization (as originally proposed by Heeger and Bergen), and using the periodic component in place of the input. One can notice slight differences between

(a) Original texture (b) Periodic component of (a) (c) Smooth component of (a)

(d) Original texture (e) Periodic component of (d) (f) Smooth component of (d)

Figure 10: **Example of "periodic + smooth" decomposition**. The mean of the input is added to the smooth component for visualization purpose. Case (a): the periodic component is close to the original texture. Case (d): the periodic component differs from the original texture because of the gradation (that is noticeable in the smooth component).

these different results. Observe that with the wood texture example, only the third solution does not suffer from noticeable artifacts. More results are shown in the experiments section (see Figure 16).

The blotchy artifact that can be noticed in the wood texture of Figure 11 is inherent to the Heeger-Bergen algorithm and it can be explained as follows: the blotches correspond to the low frequency residual of the pyramid. If the input texture has a non-negligible low frequency residual, then the output texture also has a non-negligible low frequency residual (since they share the same histogram) which results in a component that corresponds to a filtered white noise with a filter that has the size of the smallest scale ($P = 4$ in this example). Since computing the periodic component attenuates the low-frequency of the texture, when using the periodic component one attenuates the low-frequency blotchy artifact. Another (more costly) solution to restrict this artifact is to increase the number of scales, provided that the size of the images are divisible by a large power of 2 (see Figure 15).

3 Experiments

In this section, we present several experiments illustrating the influence of the algorithm parameters. Let us recall that these parameters are:

- The number of iterations N_{iter} of the synthesis algorithm.

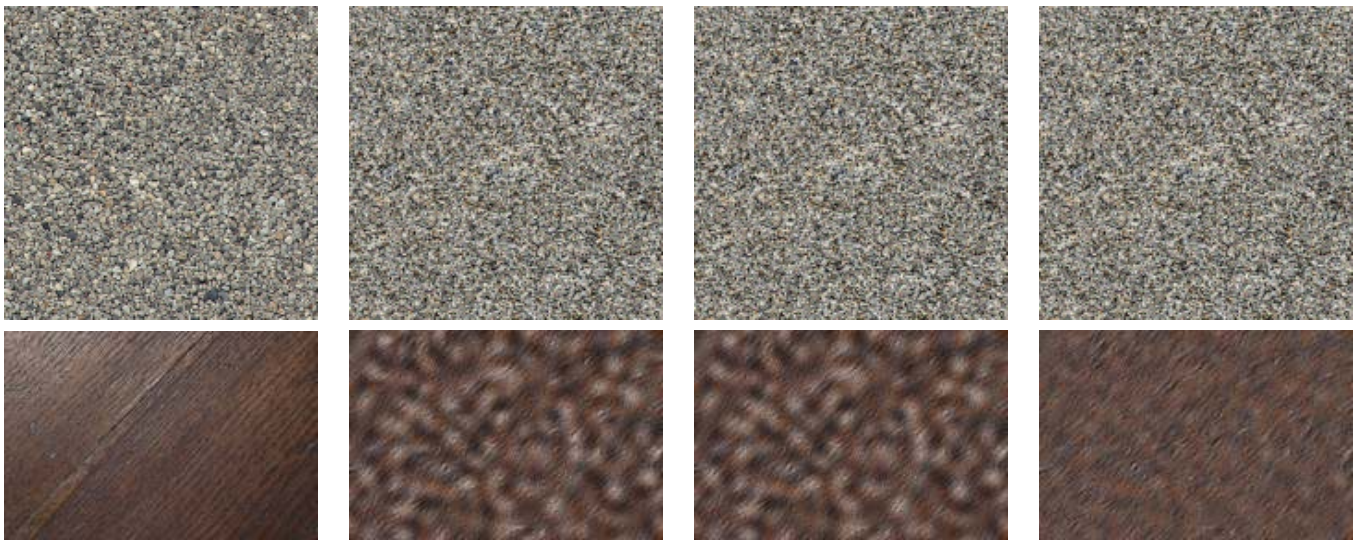


Figure 11: Comparison between the textures synthesized with different edge handling. From left to right: Original image, output without edge handling, output with mirror symmetrization at the border, and output obtained by replacing the input by its periodic component. Note that for the wood texture, only the third solution is acceptable.

- The number of scales P of the steerable pyramid decomposition.
- The number of orientations Q of the steerable pyramid decomposition.
- The edge handling option.

In the experiments that follow one varies one parameter while letting the other ones fixed to their default values. These default values are $N_{iter} = 5$, $P = 4$, $Q = 4$, and edge handling using the periodic component. For each comparative experiment, the same Gaussian white noise image is used for the initialization in order to facilitate the visual comparison of the output textures.

3.1 Influence of the Number of Iterations

As illustrated by Figure 12, the empirical convergence of the output texture is quite fast, and for most textures five iterations are enough. However for certain cases with peculiar orientation or geometry, the output quality slightly improves when increasing the number of iterations.

Let us note that, contrary to what is stated in the original article [8], we did not observe any artifact when using a large number of iterations (e.g. 100), as illustrated by Figure 13. This may be explained by the fact that our implementation makes use of perfect filters for the steerable pyramid and that we do not make any quantification for the histogram matching procedure.

Even though we did not investigate in this direction, Figure 13 may suggest that the sequence of texture images obtained at each iteration converges to a limit texture image: Indeed, when using the same input noise, there is nearly no difference between the output texture after 20, 50 or 100 iterations. To the best of our knowledge there is no theoretical proof of this convergence, and we believe that it would be a valuable theoretical justification for the whole procedure.

3.2 Influence of the Number of Orientations

Increasing the number of analyzed orientations often slightly improves the results (even if the texture has no dominant orientations, see Figure 14). However, when the sample is isotropic this parameter

Figure 12: **Influence of the number of iterations**. From left to right: original image, result with $N_{iter} = 1$, 5 and 10. For the first examples the number of iterations has a noticeable influence while it is not the case for the last one.

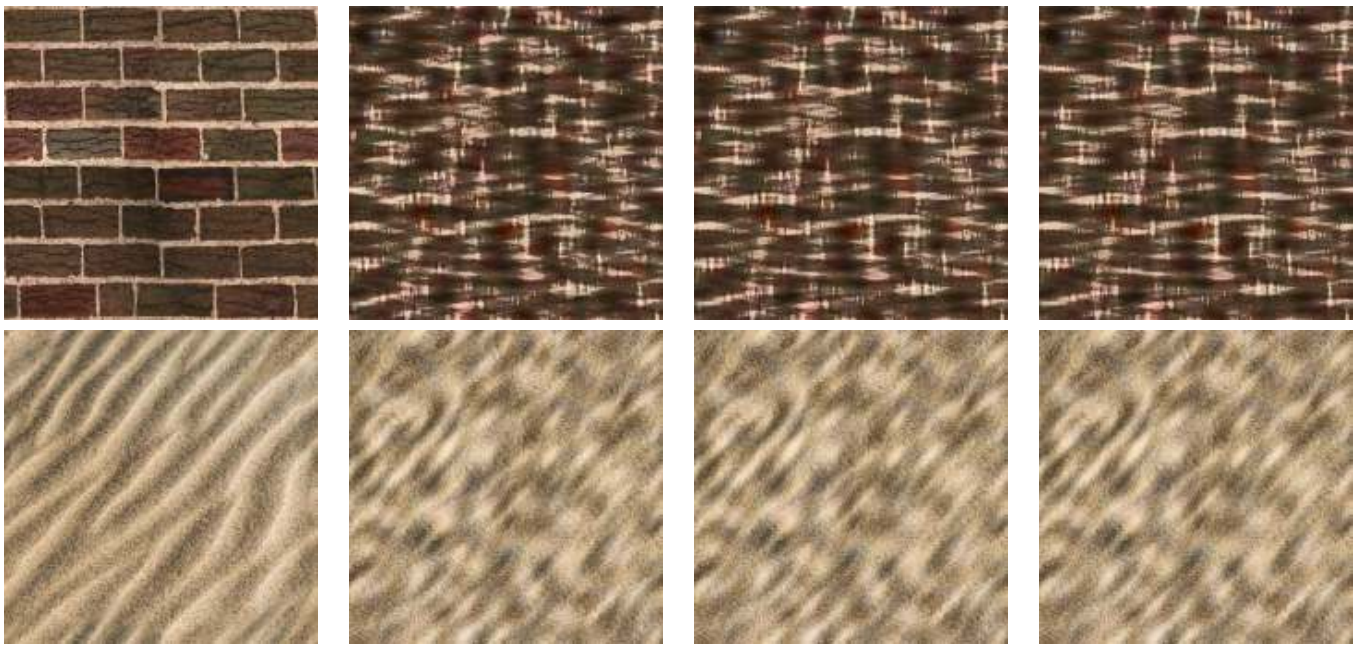


Figure 13: **Large number of iterations**. From left to right: original image, result with $N_{iter} = 20$, 50 and 100. Contrary to what is stated in [8], one does not observe any artifact when using a large number of iterations.

has no noticeable effects, and for most cases four orientations are enough.

3.3 Influence of the Number of Scales

As illustrated by Figure 15, the number of pyramid scales is the more influential parameter: using the highest value of this parameter permits to take into account all scales of the texture. Besides, using a large number of scales enables to eliminate the low frequency blotchy artifact discussed in Section 2.5. Nevertheless, for textures made of incoherent small patterns the result only weakly depends on the pyramid depth (see bottom of Figure 15).

3.4 Influence of the Edge Handling

As discussed in Section 2.5, the way of handling the edge discontinuity of the sample may have an important impact on the results of the algorithm. Figure 16 completes Figure 11. One can observe that using the periodic component gives better results for all the examples. Hence using the periodic component enables to limit the low frequency blotchy artifact without using a large number of scales.

4 Conclusion

In this paper, we have described in detail the Heeger-Bergen texture synthesis algorithm [8]. In addition of mathematical and algorithmic clarifications we proposed a minor improvement regarding edge-handling in the analysis step by using the "periodic+smooth" decomposition [13]. Numerous numerical experiments illustrate the influence of the different parameters of the algorithm and demonstrate that the proposed ANSI C implementation reproduces the result quality of the original paper [8].

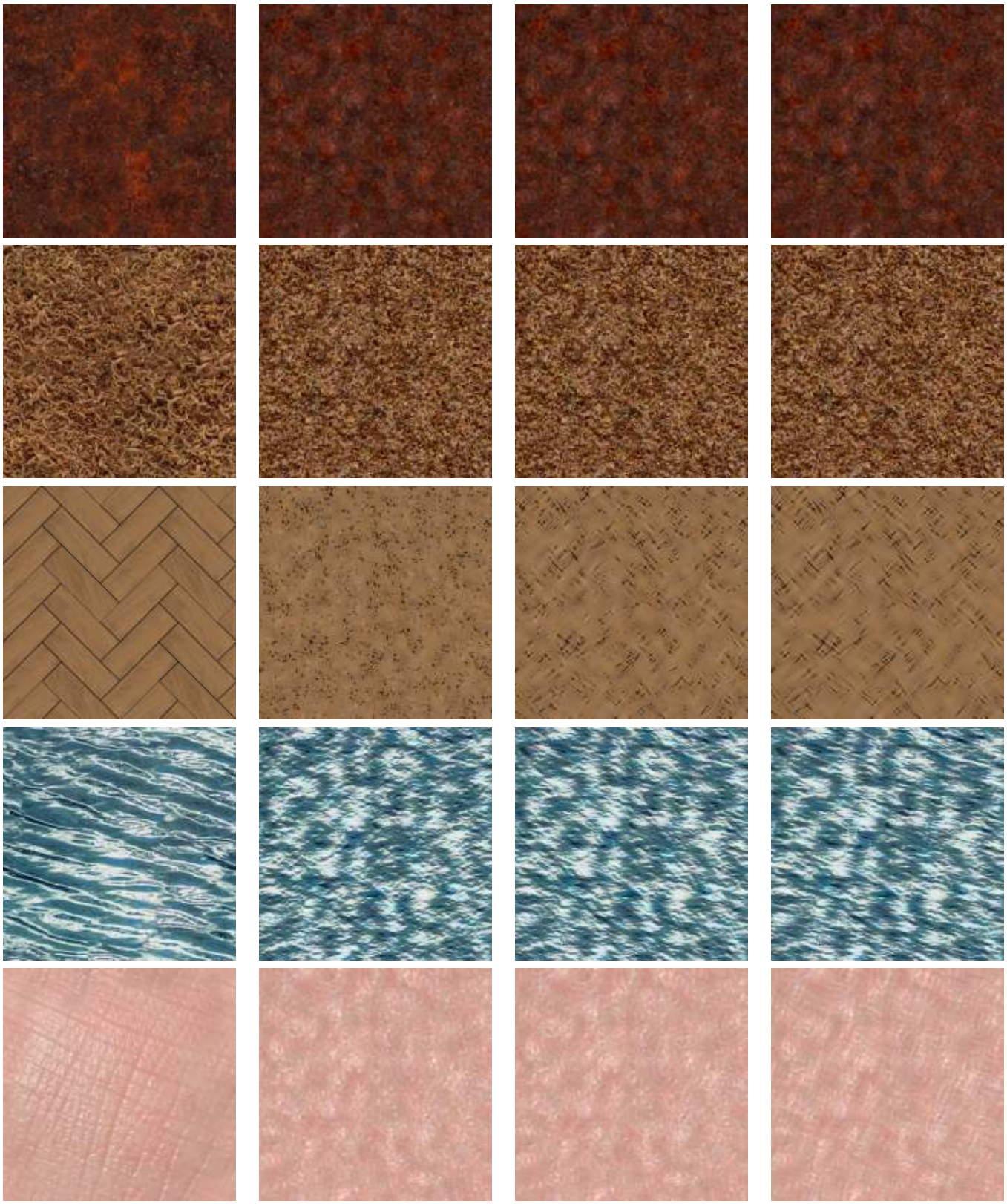


Figure 14: **Influence of the number of orientations.** From left to right: original image, result with $Q = 2$, 4 and 8 (and default values $N_{iter} = 5$ and $P = 4$).

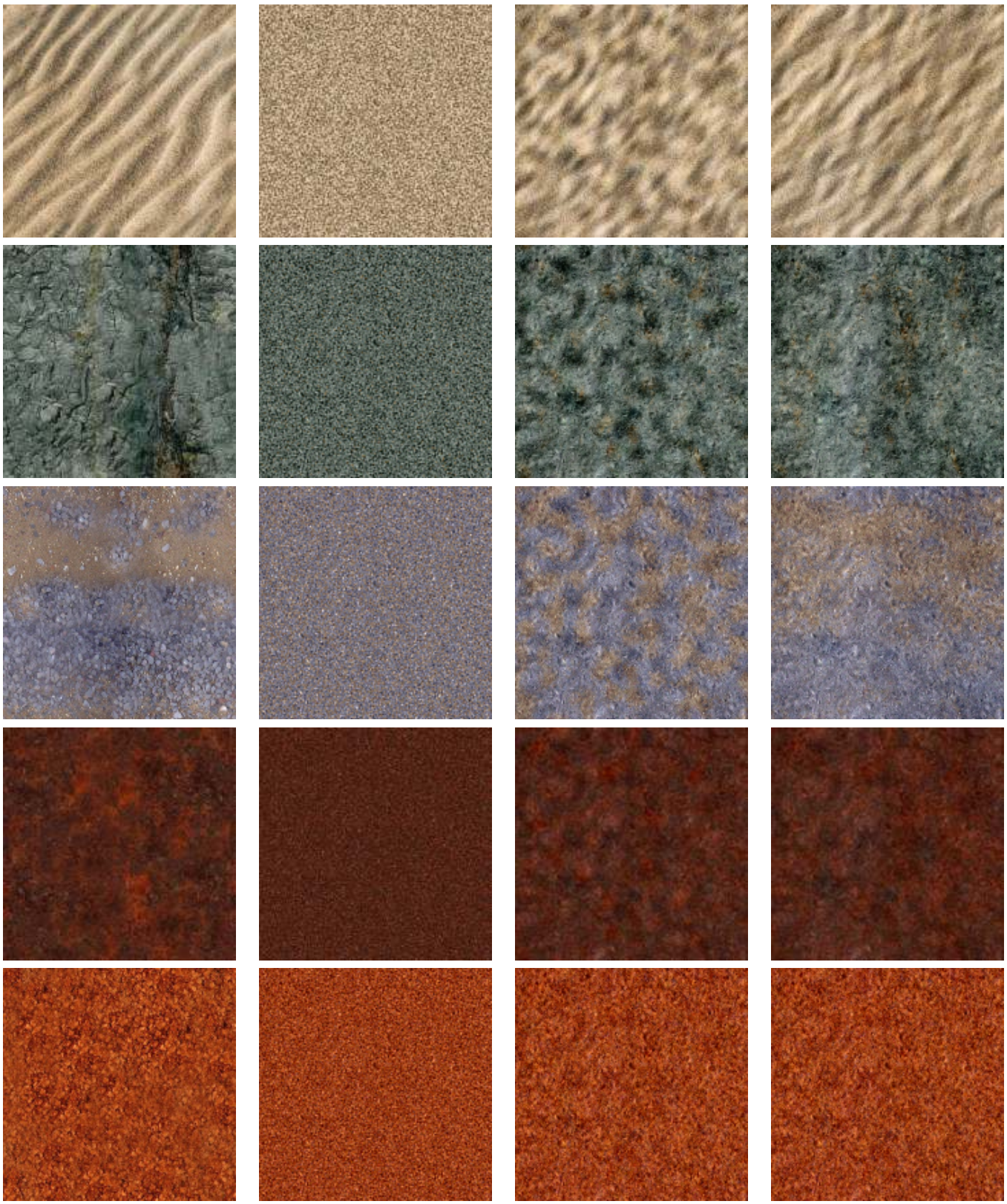


Figure 15: **Influence of the number of scales.** From left to right: original image, result with $P = 1$, 4 and 8 (and default values $N_{iter} = 5$ and $Q = 4$).

A Periodic Component Computation

We recall in this appendix how the periodic component of an image u is computed [13]. The above description as well as the corresponding C code are from IPOL paper [5].

Figure 16: **Influence of the edge handling.** From left to right: Original image, output without edge handling, output with mirror symmetrization at the border, and output obtained by replacing the input by its periodic component.

Given a discrete image u, the periodic component p of u is the unique solution of the problem

$$\begin{cases} \Delta p = \Delta_i u, \\ \text{mean}(p) = \text{mean}(u), \end{cases}$$

where Δ is the usual discrete periodic Laplacian (each point has four neighbors) and Δ_i is the discrete Laplacian in the interior of the image domain (points at the border of the image have only three or two neighbors).

Relying on this characterization, the periodic component p of u is computed using the classic FFT-based Poisson solver. More precisely p is computed by the following procedure:

1. Compute the discrete Laplacian $\Delta_i u$ of u.
2. Compute the DFT $\widehat{\Delta_i u}$ of $\Delta_i u$.
3. Compute the DFT $\hat{p}$ of p by inverting the discrete periodic Laplacian:

$$\begin{cases} \hat{p}_{m,n} = \left(4 - 2\cos\left(\dfrac{2m\pi}{M}\right) - 2\cos\left(\dfrac{2n\pi}{N}\right)\right)^{-1} \widehat{\Delta_i u}_{m,n}, & \text{if } (m,n) \in \hat{\Omega}_{M,N} \setminus \{(0,0)\} \\ \hat{p}_{0,0} = \displaystyle\sum_{k=0}^{M-1}\sum_{l=0}^{N-1} u_{k,l} & \text{if } (m,n) = (0,0) \end{cases}$$

4. Compute p by inverse DFT.

Image Credits

Most of textures come from this website: http://www.lemog.fr/lemog_textures/ and are copyright free. A few textures are personal pictures of one of the authors.

References

[1] A. Bevilacqua and P. Azzari, *A high performance exact histogram specification algorithm*, in Proceedings of the 14th International Conference on Image Analysis and Processing, 2007, pp. 623 – 628. http://dx.doi.org/10.1109/ICIAP.2007.4362846.

[2] D. Coltuc, P. Bolon, and J.-M. Chassery, *Exact histogram specification*, IEEE Transactions on Image Processing, 15 (2006), pp. 1143 – 1152. http://dx.doi.org/10.1109/TIP.2005.864170.

[3] A.A. Efros and W.T. Freeman, *Image quilting for texture synthesis and transfer*, in Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH), ACM, 2001, pp. 341–346. http://dx.doi.org/10.1145/383259.383296.

[4] A.A. Efros and T.K. Leung, *Texture synthesis by non-parametric sampling*, in Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999., vol. 2, IEEE, 1999, pp. 1033–1038. http://dx.doi.org/10.1109/ICCV.1999.790383.

[5] B. Galerne, Y. Gousseau, and J.-M. Morel, *Micro-texture synthesis by phase randomization*, Image Processing On Line, (2011). http://dx.doi.org/10.5201/ipol.2011.ggm_rpn.

[6] ——, *Random phase textures: Theory and synthesis*, IEEE Transactions on Image Processing, 20 (2011), pp. 257 – 267. http://dx.doi.org/10.1109/TIP.2010.2052822.

[7] B. Galerne, A. Lagae, S. Lefebvre, and G. Drettakis, *Gabor noise by example*, ACM Transactions on Graphics, 31 (2012), pp. 73:1–73:9. http://dx.doi.org/10.1145/2185520.2185569.

[8] D.J. Heeger and J.R. Bergen, *Pyramid-based texture analysis/synthesis*, in Proceedings of the 22nd annual conference on Computer graphics and interactive techniques (SIGGRAPH), ACM, 1995, pp. 229–238. http://dx.doi.org/10.1145/218380.218446.

[9] V. Kwatra, A.Schödl, I. Essa, G. Turk, and A. Bobick, *Graphcut textures: image and video synthesis using graph cuts*, in Proceedings of the 30th annual conference on Computer graphics and interactive techniques (SIGGRAPH), ACM Press, 2003, pp. 277–286. http://dx.doi.org/10.1145/1201775.882264.

[10] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D.S. Ebert, J.P. Lewis, K. Perlin, and M. Zwicker, *A survey of procedural noise functions*, Computer Graphics Forum, 29 (2010), pp. 2579–2600. http://dx.doi.org/10.1111/j.1467-8659.2010.01827.x.

[11] A. Lagae, P. Vangorp, T. Lenaerts, and P. Dutré, *Procedural isotropic stochastic textures by example*, Computers & Graphics (Special issue on Procedural Methods in Computer Graphics), (2010). http://dx.doi.org/10.1016/j.cag.2010.05.004.

[12] J.-L. Lisani, A. Buades, and J.-M. Morel, *Image color cube dimensional filtering and visualization*, Image Processing On Line, (2011). http://dx.doi.org/10.5201/ipol.2011.blm-cdf.

[13] L. Moisan, *Periodic plus smooth image decomposition*, Journal of Mathematical Imaging and Vision, 39 (2011), pp. 161–179. http://dx.doi.org/10.1007/s10851-010-0227-1.

[14] G. Peyré, *Texture synthesis with grouplets*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 4 (2010), pp. 733–746. http://dx.doi.org/10.1109/TPAMI.2009.54.

[15] J. Portilla and E.P. Simoncelli, *A parametric texture model based on joint statistics of complex wavelet coefficients*, International Journal of Computer Vision, 40 (2000), pp. 49–70. http://dx.doi.org/10.1023/A:1026553619983.

[16] J. Portilla, V. Strela, M.J. Wainwright, and E.P. Simoncelli, *Image denoising using scale mixtures of Gaussians in the wavelet domain*, IEEE Transactions on Image Processing, 12 (2003), pp. 1338 – 1351. http://dx.doi.org/10.1109/TIP.2003.818640.

[17] J. Rabin, G. Peyré, J. Delon, and M. Bernot, *Wasserstein barycenter and its application to texture mixing*, in Scale Space and Variational Methods in Computer Vision, vol. 6667 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2012, pp. 435–446. http://dx.doi.org/10.1007/978-3-642-24785-9_37.

[18] E. P. Simoncelli and W. T Freeman, *The steerable pyramid: A flexible architecture for multi-scale derivative computation*, in Proceedings of the International Conference on Image Processing, vol. 3, IEEE, 1995, pp. 444–447. http://dx.doi.org/10.1109/ICIP.1995.537667.

[19] E. P. Simoncelli, W. T. Freeman, E. H. Adelson, and D. J. Heeger, *Shiftable multiscale transforms*, IEEE Transactions on Information Theory, 38 (1992), pp. 587–607. http://dx.doi.org/10.1109/18.119725.

[20] G. Tartavel, Y. Gousseau, and G. Peyré, *Variational texture synthesis with sparsity and spectrum constraints*, Journal of Mathematical Imaging and Vision, (2014, accepted).

[21] C. Villani, *Optimal Transport. old and new*, Springer-Verlag, 2008. Grundlehren Der Mathematischen Wissenschaften, ISBN 3540710493.

[22] L.Y. Wei and M. Levoy, *Fast texture synthesis using tree-structured vector quantization*, in Proceedings of the 27th annual conference on Computer graphics and interactive techniques (SIGGRAPH), ACM Press, 2000, pp. 479–488. http://dx.doi.org/10.1145/344779.345009.

[23] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk, *State of the art in example-based texture synthesis*, in Eurographics 2009, State of the Art Report, EG-STAR, Eurographics Association, 2009.