Published in Image Processing On Line on 2013–10–23.
Submitted on 2013–01–11, accepted on 2013–05–28.
ISSN 2105–1232 © 2013 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2013.59

Exemplar-Based Texture Synthesis: the Efros–Leung Algorithm

Cecilia Aguerrebere[1], Yann Gousseau[2], Guillaume Tartavel[3]

[1] IIE, UdelaR, Uruguay and Télécom ParisTech, LTCI CNRS, France (aguerreb@telecom-paristech.fr)
[2] Télécom ParisTech, LTCI CNRS, France (gousseau@telecom-paristech.fr)
[3] Télécom ParisTech, LTCI CNRS, France (tartavel@telecom-paristech.fr)

Communicated by Julie Digne *Demo edited by* Cecilia Aguerrebere

Abstract

Exemplar-based texture synthesis aims at creating, from an input sample, new texture images that are visually similar to the input, but are not plain copy of it. The Efros–Leung algorithm is one of the most celebrated approaches to this problem. It relies on a Markov assumption and generates new textures in a non-parametric way, directly sampling new values from the input sample.

In this paper, we provide a detailed analysis and implementation of this algorithm. The code closely follows the algorithm description from the original paper. It also includes a PCA-based acceleration of the method, yielding results that are generally visually indistinguishable from the original results.

To the best of our knowledge, this is the first publicly available implementation of this algorithm running in acceptable time. Even though numerous improvements have been proposed since this seminal work, we believe it is of interest to provide an easy way to test the initial approach from Efros and Leung. In particular, we provide the user with a graphical illustration of the innovation capacity of the algorithm. Experimentation often shows that the path between *verbatim copy* of the exemplar and *garbage growing* is somewhat narrow, and that in most favorable cases the algorithm produces new texture images by stitching together entire regions from the exemplar.

Source Code

The ANSI C source code, the code documentation, and the online demo are accessible from the IPOL web page of this article here[1].

Keywords: efros leung, texture synthesis, patches, exemplar-based.

[1] https://doi.org/10.5201/ipol.2013.59

1 Introduction

Until the mid 90's, the most successful texture synthesis algorithms relied on relatively ad hoc Markov random field models, a well known example of which may be found in the paper by Cross and Jain [4]. In 1995, a first breakthrough occurred with the work of Heeger and Bergen [11][2], who proposed to synthesize new textures by prescribing the marginals of the different sub-bands of a wavelet decomposition on the one hand, and of the color content on the other hand. This approach was followed by more elaborate ones suggesting to perform synthesis by adding scale dependencies [5] or by enforcing other statistical constraints, notably second order dependency between wavelet coefficients at different positions, scales and orientations [15]. An alternative approach, well suited for non or weakly structured textures, consists in enforcing the Fourier power spectrum of images using a *random phase* approach [9][3].

A second major advance was provided in 1999 by the work of Efros and Leung [8], introducing a non-parametric approach to Markov modeling in the context of texture synthesis. As we will see in detail in the present contribution, the basic idea of this work is to use an exemplar image as a source from which pixel values are chosen to perform the synthesis, one pixel at a time, depending on the agreement of their neighborhood with the already synthesized part of the output image. This approach yields visually striking synthesis results, even on highly structured texture images. The method may be seen as an automated way to generate new images by a copy-paste procedure. A similar but faster approach was also independently proposed by Wei and Levoy [17]. The original Efros–Leung algorithm has been followed by a large body of contributions. In particular, speeded-up approaches suggest to synthesize new images by pasting patches instead of single pixels [6, 14] or rely on parallelization procedures [13]. A complete state of the art of such *copy-paste* approaches is beyond the scope of this paper. An overview may be found in the paper by Wei et al. [16]. Let us also mention that this work by Efros and Leung has been an inspiration for many works outside the field of texture synthesis. The most striking example of such works is the NL-means approach to image denoising [3], that in turn has triggered the explosion of *patch-based* methods for image restoration.

Despite being highly influential, the original algorithm of Efros and Leung has not been widely tested. This is mostly due to its computational complexity, but it is somehow surprising that, to the best of our knowledge, no correct implementation of the algorithm is available that runs within acceptable time (at least enabling experimentation). The goal of the demo accompanying this contribution is to provide such an implementation. We believe that despite the fact that this algorithm is outdated, in particular concerning its computational complexity, such a demo is useful because the basic principles of the algorithm are still widely used in practical and fast implementations [16].

Experiments show that, even though it often produces visually impressive results, the Efros–Leung method suffers from two important drawbacks:

1. It often yields *verbatim copy* of the input exemplar.

2. It may *grow garbage*, that is, produce inconsistent results from which it cannot recover.

These two points are mentioned (in the same terms) in the original paper [8] as possible failure cases. They are in fact very common in experiments. In many cases, it is actually difficult or impossible to tune the *tolerance* parameter (that we will detail below) to avoid both drawbacks 1 and 2 above. In particular, verbatim copying happens very frequently, as it may be seen by looking at experiments from the original paper. Notable exceptions are binary and small scale textures, for

[2] An online demo for the Heeger–Bergen method is available here [1]
[3] An online demo for the Galerne et al. method is available here [10]

which real innovation is permitted by the algorithm. In order to allow for a direct investigation of the verbatim copying tendency of the algorithm, we provide in the demo a graphical illustration of the synthesis process enabling one to identify parts (groups of pixels) of the sample image that have been copied. Often, the resulting synthesized image is a patchwork of regions from the exemplar. This somehow justifies approaches in which the synthesis is explicitly performed by stitching pieces from the sample image [18, 12].

2 Algorithm

2.1 Informal Description

Informally, the original Efros–Leung algorithm works as follows. We consider an original image A and an image B to be synthesized. We assume that a tiny portion of the image, thereafter called a *seed*, is already known. The synthesis then proceeds iteratively, one pixel at a time, growing layers outwards from this initial seed. Given a pixel p to be synthesized, we consider a $n \times n$ squared neighborhood around it, thereafter called a *patch*. The portion of this patch made of pixels with known values (pixels that have already been synthesized in previous steps or pixels from the initial seed) is called *known patch*. We then look for all pixels in the original image A having a similar known patch as the pixel p and draw one of them at random. Eventually, we affect the color value of the central pixel of this randomly chosen patch to p.

2.2 Detailed Description

We now describe the synthesis algorithm in more detail. Again, A is the sample image and B the one to be synthesized. Images are defined in a rectangular subset of $\mathbb{Z}^2$ and pixels are defined as elements of these subsets (spatial positions). For a pixel p in either image, we write $\mathcal{N}(p)$ for the $n \times n$ patch around p (that is, the set of all pixel values in the $n \times n$ neighborhood of p), where n is an odd integer.

Patch distance For two pixels p' in A and p in B, we define a Gaussian-weighted distance between the corresponding neighborhood as

$$d(\mathcal{N}(p), \mathcal{N}(p')) = \frac{1}{\sum_{i \in \mathcal{N}_0} G_\sigma(i)} \sum_{i \in \mathcal{N}_0} \left(A(p'+i) - B(p+i)\right)^2 G_\sigma(i), \tag{1}$$

where G_σ is a Gaussian with standard deviation σ and $\mathcal{N}_0$ is an $n \times n$ squared window centered on the origin.

Single pixel synthesis Let us assume for the moment that for a given pixel p in the image B, all the values in $\mathcal{N}(p)$ are known. We then compute:

- The minimum distance from $\mathcal{N}(p)$ to all the patches in A,

$$d_m(p) = \min_{p' \in A} d(\mathcal{N}(p), \mathcal{N}(p')). \tag{2}$$

- The set of pixels similar to p, for a given tolerance parameter ε:

$$S_\varepsilon(p) = \left\{p' \in A : d(\mathcal{N}(p), \mathcal{N}(p')) \leq (1+\varepsilon) d_m(p)\right\}. \tag{3}$$

In order to synthesize the value at p, we then draw at random a pixel from S_ε, with equal weights for all the pixels.

Sequential synthesis It is not feasible to assume that all the neighbors of a given pixel are known, and the previous single-pixel procedure is modified as follows. First, the distances $d(\mathcal{N}(p), \mathcal{N}(p'))$ are computed only for the known neighbors of p (and the normalization factor is adapted to the number of known pixels). Second, the synthesis is performed iteratively, in layers, starting from an initial $k \times k$ region that is assumed to be known, called seed. In the original algorithm and from now on, this seed is assumed to be a 3×3 excerpt from the original image A. For the first pixel, the only known region on which d is computed is the seed. Then, iteratively, synthesized pixels are added to the known region and used to compute d.

2.3 Pseudo-code

A pseudo-code for the implementation of the Efros–Leung algorithm used in the present demo is given in algorithm 1. This implementation follows closely the presentation of the original paper [8] and the pseudo-code given at the authors website [7]. Let us mention, however, that both presentations differ in one point which will be commented below. We have chosen to follow the original method [8]. Also, some of the steps of this pseudo-code are not unambiguously described in the original paper [8] and will be specified.

Algorithm 1: Pseudo-code of the Efros–Leung algorithm.

```
function GrowImage( SampleImage, Image, WindowSize )
while Image not filled do
    PixelList = GetUnfilledNeighbors( Image )
    for Pixel in PixelList do
        Template = GetNeighborhoodWindow( Pixel )
        BestMatches = FindMatches( Template, SampleImage)
        BestMatch = RandomPick( BestMatches )
        Pixel.value = BestMatch.value
    end
end
Return Image
```

Algorithm 2: Pseudo-code of the `FindMatches` function.

```
function FindMatches( Template, SampleImage )
ValidMask = 1s where Template is filled, 0s otherwise
GaussMask = Gaussian2D( WindowSize, Sigma )
TotWeight = sum i,j GaussMask(i,j)*ValidMask(i,j)
for i,j do
    for ii,jj do
        dist = ( Template(ii,jj) - SampleImage(i-ii,j-jj) )^2
        SSD(i,j) = SSD(i,j) + dist*ValidMask(ii,jj)*GaussMask(ii,jj)
    end
    SSD(i,j) = SSD(i,j) / TotWeight
end
PixelList = all pixels (i,j) where SSD(i,j) ≤ min(SSD)*(1+ErrThreshold)
Return PixelList
```

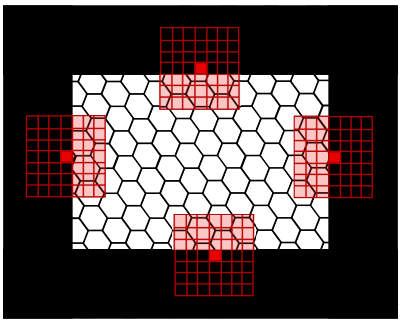


Figure 1: Since the synthesis is performed in outwards layers starting from an initial seed, most patches soon have half their pixel values known, either on the left, right, top or bottom side.

The function `GetUnfilledNeighbors` returns a list of all unfilled pixels that have filled pixels as their neighbors (the image is subtracted from its morphological dilation). The list is randomly permuted and then sorted by decreasing number of filled neighbor pixels. `GetNeigborhoodWindow` returns a window of size WindowSize around a given pixel. `RandomPick` picks an element randomly from the list. The function `FindMatches` is presented in algorithm 2.

The function `Gaussian2D` generates a two-dimensional Gaussian in a window of given size centered at the origin and with a given standard deviation (in pixels). In the original implementation the constants were set as follows: ErrThreshold = 0.1, Sigma = WindowSize/6.4.

This pseudo-code follows the original description from Efros and Leung [8]. In the pseudo-code given at the web site [7], one additional step is added. After the call to the `RandomPick` function, the value BestMatch.value is affected to Pixel.value only if the computed distance between the corresponding neighborhoods is small enough. This may improve synthesis results in some cases. However, it adds new parameters to the algorithm and can also be very time consuming. We therefore decided to implement the original algorithm [8], as described above.

3 Speed-up using PCA

General idea The original algorithm is very time-consuming: generating M pixels from a sample image of N pixels, using windows containing K pixels, requires O(MNK) operations.

Most of the computing time is spent in computing inter-patches distances. A first easy way to accelerate this computation is as follows. For a given pixel p in the synthesized image B, the computation of $d(\mathcal{N}(p), \mathcal{N}(p'))$ is stopped as soon as the threshold $(1+\varepsilon)d_m$ is reached, i.e. as soon as we know that the patch from the original image we are looking at is not in S_ε, as defined by (3).

Next, we observe that using patch coordinates in which the first coefficients are likely to be the largest, the previous threshold can be reached much faster. A principal component analysis (PCA) basis is used for this purpose.

Implementation Recall that the synthesis is performed in outwards layers, starting from an initial seed. As can be seen in figure 1, most patches soon have half their pixel values known, either on the left, right, top or bottom side.

We exploit this fact by using four PCA basis. Each basis is computed from all the top, bottom, left or right halves of the patches in the sample image. Then the distance between patches is computed, for a given pixel p, as

$$d(p,q) = d(p_{pca}, q_{pca}) + d_{extra_{pix}}, \tag{4}$$

where p_{pca} is the projection of the known half of the patch containing p onto the corresponding PCA basis and q_{pca} is an element of the corresponding PCA dictionary. Four dictionaries are created at

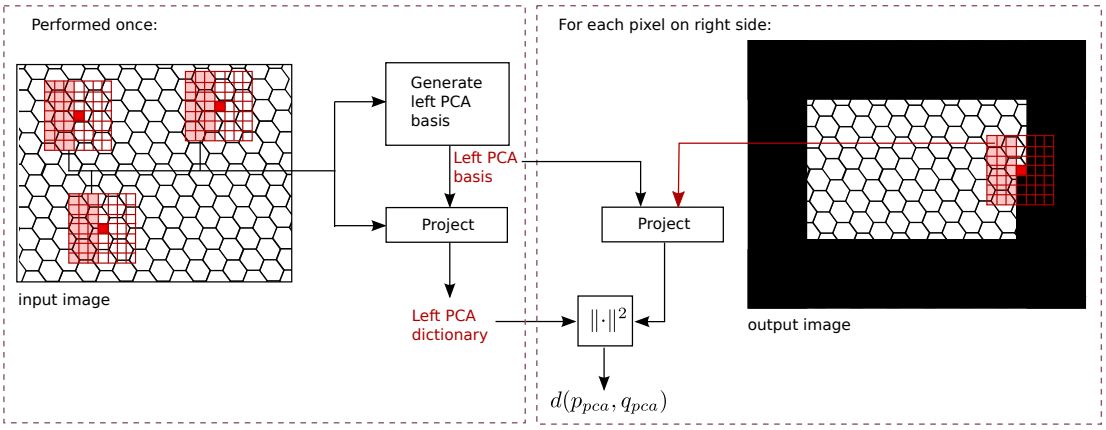


Figure 2: The *left* PCA basis is created from all the left-halves of patches in the sample image. Then the *left* dictionary is created, containing the projections of all the left-halves of patches in the sample image onto the *left* PCA basis. To fill a pixel for which the left half has already been synthesized, the known half of the corresponding patch is projected onto the *left* PCA basis and the distance to all elements of the *left* dictionary are computed. The algorithm continues as for the classical case. The same procedure is performed for the right, top and bottom side cases.

the beginning of the algorithm. For instance, the *left* PCA dictionary contains the projections of all the left-halves of patches in the sample image onto the *left* PCA basis (see figure 2). If some extra pixels are also known (those not belonging to the known half), the squared difference to the remaining known pixels d_{extra} is added to the distance computation.

Moreover, it was also observed that for most textures, the information needed for synthesis is carried by luminance and not by the color channels. Thus, another important reduction in computation time is obtained by converting the input RGB image to a gray level image. For this purpose, we perform a preliminary PCA over the RGB tridimensional space and keep only the first component. Observe that for a gray level image, the PCA speed-up is actually yielding the exact same result as the original algorithm. For color images, results are most of the time visually identical, except in rare cases (see section 6).

Finally, the computation of distances between a given patch and all the candidates is parallelized straightforwardly by splitting the dictionary into sub-parts.

Pseudo-code A pseudo-code for the implementation of the accelerated version of the Efros–Leung algorithm previously described is presented in algorithm 3.

The function `CreatePCADictionaries` returns four lists containing the PCA projections of all the half patches in the sample image, e.g. DictTop is the list of projections of all the top halves. The function `GetNeighborhoodCentral` returns the pixels of the current patch with the same column (for the case of right or left known halves) or row (for the case of top or bottom known halves) as the current pixel. Those pixels are the extra known pixels which do not belong to the known half. The functions `GetUnfilledNeighbors`, `GetNeighborhoodWindow`, `FindMatches` and `RandomPick` match those already introduced in the previous pseudo-code. The function `FindMatchesPCA` is summarized in algorithm 4.

The function `GetKnownHalf` returns the pixels corresponding to the known half of the current patch. The function `PCAProjection` returns the PCA projection of the input, in this case the current known patch half. The function `Gaussian2D` was already introduced in the previous pseudo-code.

Algorithm 3: Pseudo-code of the PCA based accelerated version of the Efros–Leung algorithm.

```
1  function GrowImageSpeedUp( SampleImage, WindowSize, dimsPCA )
2  [ DictTop, DictBottom, DictRight, DictLeft ] = CreatePCADictionaries( SampleImage,
   WindowSize, dimsPCA )
3  while Image not filled do
4      PixelList = GetUnfilledNeighbors( Image )
5      for Pixel in PixelList do
6          if Half patch already known then
               // Use PCA
7              switch Known Half do
                   // Choose the dictionary
8                  case Top
9                      DictPCA = DictTop
10                 end
11                 case Bottom
12                     DictPCA = DictBottom
13                 end
14                 case Right
15                     DictPCA = DictRight
16                 end
17                 case Left
18                     DictPCA = DictLeft
19                 end
20             end
21             ExtraKnown = GetNeighborhoodCentral( Pixel )
22             BestMatches = FindMatchesPCA( ExtraKnown, SampleImage, DictPCA )
23             BestMatch = RandomPick( BestMatches )
24             Pixel.value = BestMatch.value
25         else
               // Follow classical approach
26             Template = GetNeighborhoodWindow( Pixel )
27             BestMatches = FindMatches( Template, SampleImage )
28             BestMatch = RandomPick( BestMatches )
29             Pixel.value = BestMatch.value
30         end
31     end
32 end
33 Return Image
```

Algorithm 4: Pseudo-code of the `FindMatchesPCA` function.

```
1  function FindMatchesPCA( ExtraKnown, SampleImage, DictPCA )
2  GaussMaskPCA = sqrt( Gaussian2D( KnownHalf, Sigma ) )
3  KnownPatchHalf = GetKnownHalf( Pixel )*GaussMaskPCA
4  PatchPCA = PCAProjection( KnownPatchHalf, DictPCA )
5  ValidMask = 1s where ExtraKnown is filled, 0s otherwise
6  GaussMaskCentral = Gaussian2D( WindowCentral, Sigma )
7  for i,j do
       // For all patches
8      distLimit = infinite
9      while (ii,jj in central) AND (SSDCentral < distLimit) do
           // Compute distance between central coordinates only AND stop if
              SSDCentral is above the current distance limit
10         distCentral = ( ExtraKnown(ii,jj) - SampleImage(i-ii,j-jj) )^2
11         SSDCentral = SSDCentral + distCentral*ValidMask(ii,jj)*GaussMaskCentral(ii,jj)
12     end
13     if SSDCentral < distLimit then
14         while (k = 1 to dimsPCA) AND (SSDCentral + SSD_PCA < distLimit) do
15             distPCA = ( PatchPCA(k) - DictPCA(i,j,k) )^2
16             SSD_PCA(i,j) = SSD_PCA(i,j) + distPCA
17         end
18     end
19     SSD(i,j) = SSDCentral + SDD_PCA
20     distLimit = min(SSD)*(1+ErrThreshold)
21 end
22 PixelList = all pixels (i,j) where SSD(i,j) ≤ min(SSD)*(1+ErrThreshold)
23 Return PixelList
```

Experiment	Time (s)	
	patch width 11, sample size 128	patch width 15, sample size 512
classical	8.7	665
partial dist. comp.	**5.9**	360
partial dist. comp. + PCA (all dims)	6.5	160
partial dist. comp. + PCA (20% dims)	**5.9**	**145**

Table 1: Computation times for each of the different algorithmic options. Two cases are tested: patch size 11×11, input sample size 128×128, output size 128×128 and patch size 15×15, input sample size 512×512, output size 128×128. For a small enough input sample and patch size, the gain in performance of the accelerated versions with respect to the classical approach is not very significant. Recall that in the classical approach, the computation of distances between a given patch and all the candidates is parallelized straightforwardly by splitting the dictionary into sub-parts. On the other hand, the computation time for the accelerated versions is considerably reduced for larger input samples and patch sizes.

Computing distances using a fixed number of dimensions We observed that limiting the computation of distances on a fixed number of dimensions n generally yields results that are visually identical to the original results, for values of n in the order of 20% of the dimensionality of the patch space. This provides a further speed-up of the algorithm. In the online demo accompanying this paper, the user has two options. Either running the original algorithm (with partial distance computations) or running the computation of distances using a fixed and chosen number n of PCA dimensions. In table 1, we display computation times for the different algorithmic options presented in this section and for different sample image and patch sizes.

4 Graphical Innovation Monitoring

In order to evaluate the innovation capacity of the method and to know the exact regions where verbatim copy occurs, we use a map of pixel positions which we call *synthesis map*. Each pixel position p in the sample image A is associated to a different color from a continuous colormap, as illustrated in figure 3, second image from the left. We call the resulting image a *position map*. The *synthesis map* of the synthesized image B is then obtained by mapping each position p to the color value of the position p' (in A) of the pixel used for the synthesis. This way, regions of the synthesis map corresponding to verbatim copy are extracts from the position map and can be easily visualized.

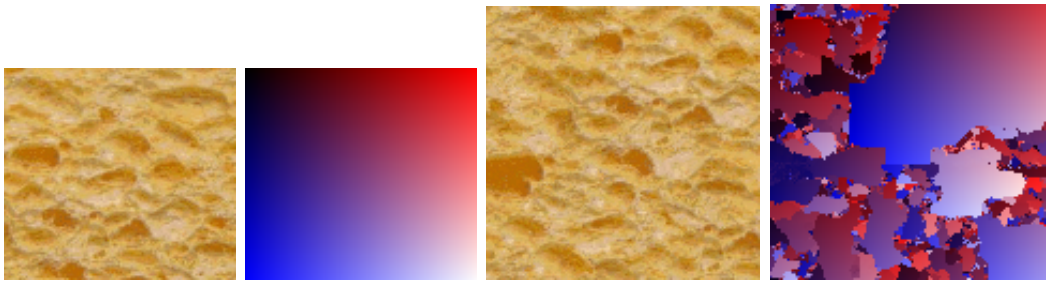


Figure 3: **Left to right:** sample image, position map, synthesized image, synthesis map. The verbatim copied regions of the sample image can be identified in the synthesis map as extracts from the position map.

5 Numerical Experiments and Parameter Setting

In this section, we perform texture synthesis experiments using the previously described method. We pay attention to both the visual aspect of the results and the way they differ from the exemplar (the innovation capacity of the method). In sections 5.2 and 5.3 we comment these aspects with respect to parameter choices. These are the patch size n and the tolerance parameter ε. In section 5.1 we briefly illustrate the relationship between the scale of the texture to be synthesized and the patch size.

Images are generated using our implementation of the classical Efros–Leung algorithm (not the accelerated implementation). For all experiments, the standard deviation of the Gaussian used for patch distances is set to $\sigma = n/6.4$ as in the Efros–Leung web page [7] and the initial seed is a randomly chosen 3×3 excerpt from the sample image. Unless specified, the tolerance parameter is set to 0.1.

5.1 Texture Scale and Patch Size

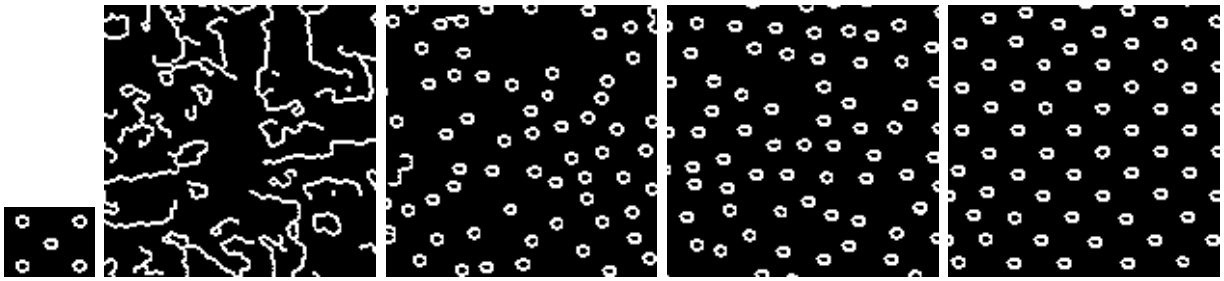


Figure 4: **Texture scale. From left to right:** sample image, results obtained with patch widths 5, 11, 15, and 23. The original image is from the Efros-Leung paper [8] and results are obtained with our implementation. A patch size of 5 is too small to recreate the circles of diameter 10. A patch size of 11 can represent the circles but not their spatial pattern. Increasing the patch size to 23 gives a correct result.

As illustrated in the original article [8], setting the patch size is critical in order to correctly synthesize a texture. Patches must be large enough to contain the texture patterns to be synthesized. The example presented in figure 4 clearly illustrates this idea. This example is taken from the original work by Efros and Leung but the presented images are generated with our implementation of the method. This synthesis algorithm is actually very well suited to such binary textures.

5.2 Innovation Capacity and Patch Size

When the patch size increases, it becomes more difficult to find "good" candidates. More precisely, for a given ε, the size of $S_\varepsilon(p)$ decreases as n increases. Hence, increasing the patch size results in increased verbatim copy. This section graphically illustrates this fact, while the next one shows that increasing ε does not solve the problem.

The sample image for the first test case is displayed in the first column of figure 5. Columns two to five show results[4] with patch sizes of 3, 5, 9 and 17, respectively. The sample image is of size 79×79 pixels and the synthesized images are of size 128×128 pixels. Good results are obtained

[4]For all the examples, results obtained in one run of the algorithm are presented. Nevertheless, the tests were repeated several times and the presented results are representative of the obtained results.

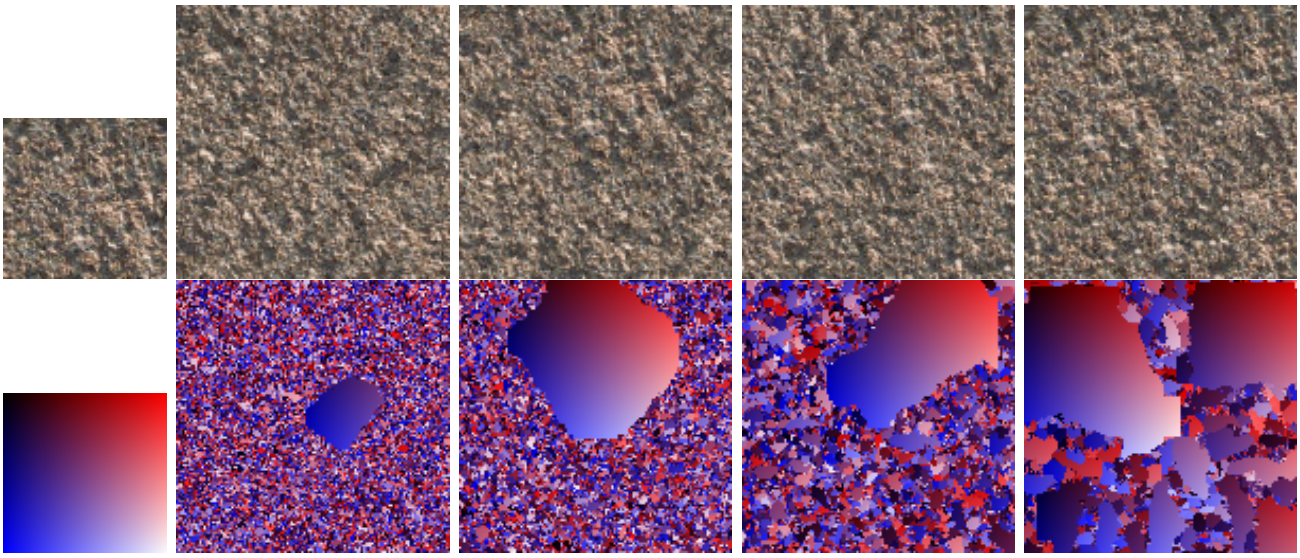


Figure 5: **Patch size variation. First row (from left to right):** sample image, results obtained with patch sizes 3, 5, 9, and 17. **Second row:** Corresponding synthesis maps. Good results are obtained in all cases. The innovation capacity decreases with increasing n, as can be verified in the decreasingly noisy aspect of the synthesis maps.

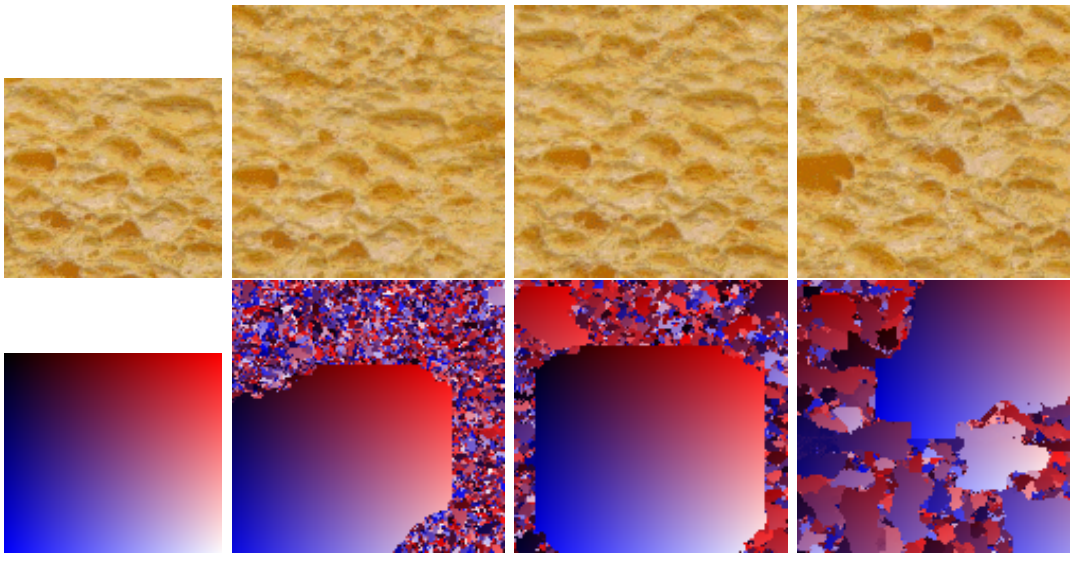


Figure 6: **Patch size variation. First row (from left to right):** sample image, results obtained with patch sizes 5, 9, and 17. **Second row:** Corresponding synthesis maps. Unlike the previous example (figure 5), the regions of the output images which are not verbatim copied do not faithfully represent the sample image.

in all cases, with a slight improvement when increasing n. Regarding the innovation capacity of the algorithm, it decreases with increasing n. The synthesis maps shown in the second row of figure 5 are much noisier for the n values 3 and 5 than for values 9 and 17, illustrating the fact that larger regions are verbatim copied for larger n values.

A second test case is displayed with the same layout in figure 6. Once again, it can be verified that results improve with increasing n, at the price of a decrease of the innovation capacity of the algorithm. Unlike the previous example, the regions of the output images that are not verbatim copied do not faithfully represent the sample image. Good results are obtained for $n = 17$, but most of the synthesized image is composed of verbatim copied regions. For such a large patch size, the set of good candidates is very restricted and copy is almost the only choice. In order to have a good result with less copied regions a much larger sample image would be needed.

Figure 7 shows the results obtained with other test examples. Conclusions are similar as for the first example. The innovation capacity of the algorithm is quite restricted on these examples.

5.3 Influence of the Tolerance Parameter

The tolerance parameter ε controls the *quality* of the patch candidates in $S_\varepsilon(p)$. The larger ε is, more patch candidates can differ from the best patch candidate. On the one hand, large tolerance values are prone to *grow garbage*, that is, produce inconsistent results from which the algorithm cannot recover. On the other hand, small tolerance values are prone to reduce the innovation capacity of the method.

The sample image for the first test case is displayed in the first column of figure 8. Columns two to five show examples of the obtained results with the tolerance parameter set to 0.05, 0.1, 0.3, and 0.5 respectively. The sample image is of size 99×99 pixels and the synthesized images are of size 128×128 pixels. The patch size is set to 9, the minimum tested value yielding reasonable results for this example. We are interested in the minimum patch size capable of synthesizing the texture, in order to limit verbatim copy as much as possible (see section 5.2) and in order to evaluate the copy level caused by the variation of the tolerance parameter.

As expected, the innovation capacity of the method increases with ε. This can be verified in the increasingly noisy aspect of the synthesis maps. For small ε values, results are visually good but essentially made of pieces from the sample image stitched together. Results are very noisy for large ε. The maximum tested value that gives good results is $\varepsilon = 0.3$. Nevertheless, the proportion of copied regions is not negligible and it can be seen that most of the regions that give the *realistic* aspect to the synthesized texture (darker zones) correspond to verbatim copied regions of the sample image (see figure 8). For this example, the best compromise between innovation and quality of the results is obtained for a patch size of 9 and a tolerance parameter of 0.1.

Another example showing a similar behavior can be seen in figure 9. A third example is shown in figure 10. For this case, the patch size is set to 5, again the minimum tested value yielding reasonable but not copied results. The patch size being small, the copy level of the results is low. This can be verified in the noisy aspect of the synthesis maps. Results remain visually satisfying for large ε values. This contrasts with the previous examples and is mostly due to the noisy nature of the texture.

6 Reproducibility of the Original Results

In the present section we show a comparison of some of the results obtained by Efros et Leung [8] and those obtained using our implementations of the algorithm, both the classical and the accelerated version. Figures 11 and 12 present these examples. Note that the results obtained using the accel-

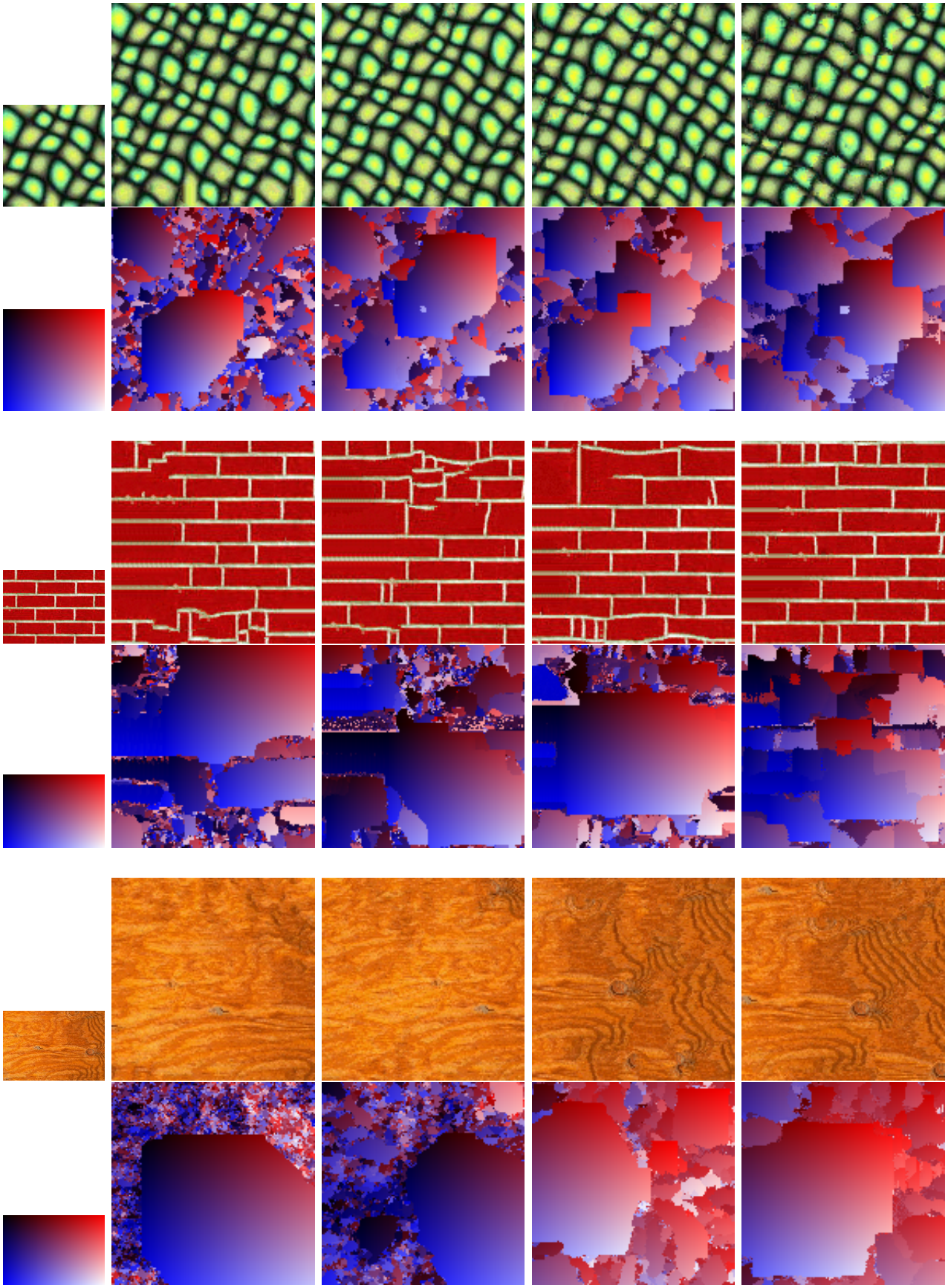


Figure 7: **Patch size variation. First/Second row (from left to right):** sample image, results obtained with patch sizes 9, 17, 25, and 33 and the corresponding synthesis maps. **Third/Fourth row:** same layout (sizes 9, 13, 17, and 25). **Fifth/Sixth row:** same layout (sizes 5, 9, 17, and 25). The innovation capacity of the algorithm is quite restricted on these textures.

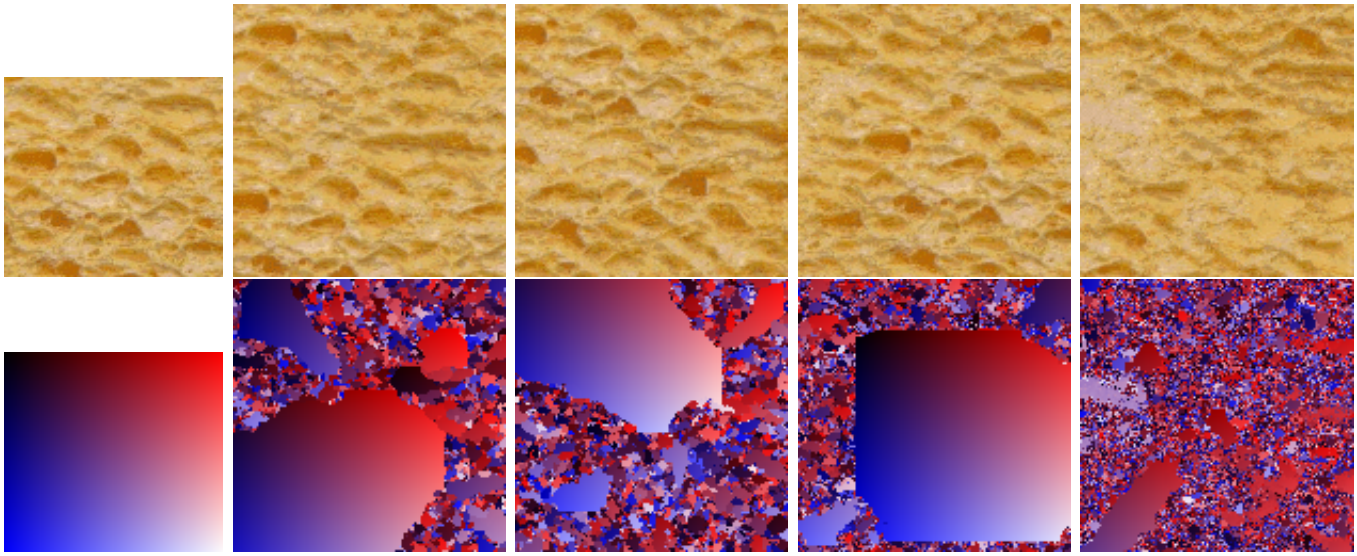


Figure 8: **Tolerance parameter variation. First row (from left to right):** sample image, results obtained with ε taking values 0.05, 0.1, 0.3, and 0.5. The patch size is set to 9. **Second row:** Corresponding synthesis maps. The innovation capacity of the method increases with ε, as can be verified in the increasingly noisy aspect of the synthesis maps. Results are good for small ε values, but are essentially made of small regions from the exemplar that are stitched together. Results are very noisy for large ε.

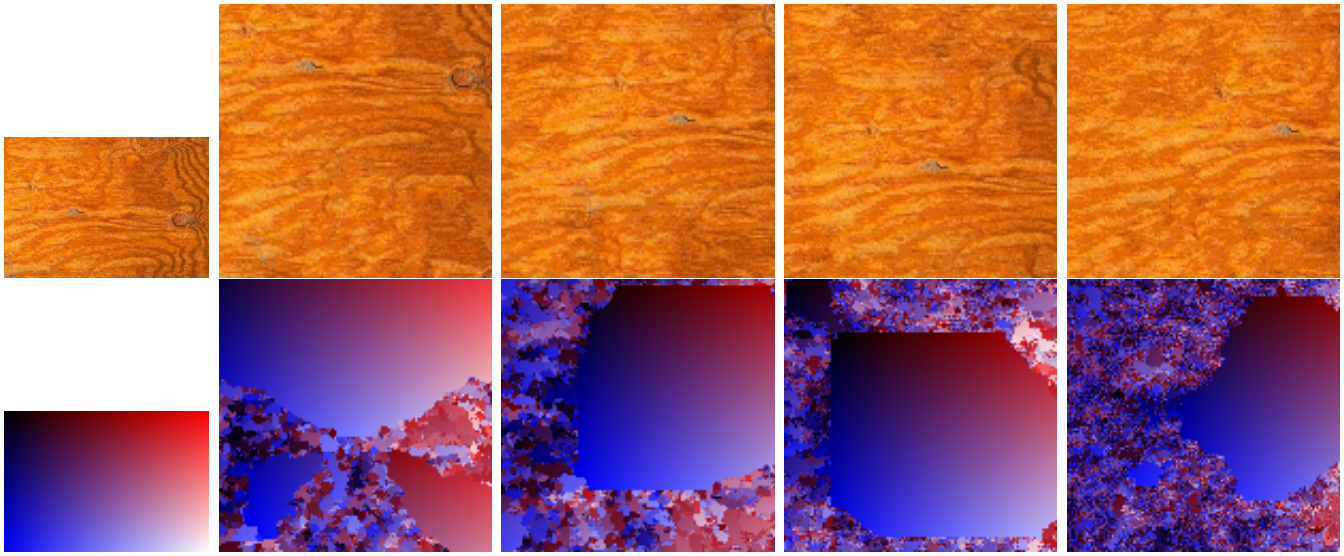


Figure 9: **Tolerance parameter variation. First row (from left to right):** sample image, results obtained with ε taking values 0.05, 0.1, 0.3, and 0.5. The patch size is set to 9. **Second row:** Corresponding synthesis maps. As in the previous example, the innovation capacity of the method increases with ε.

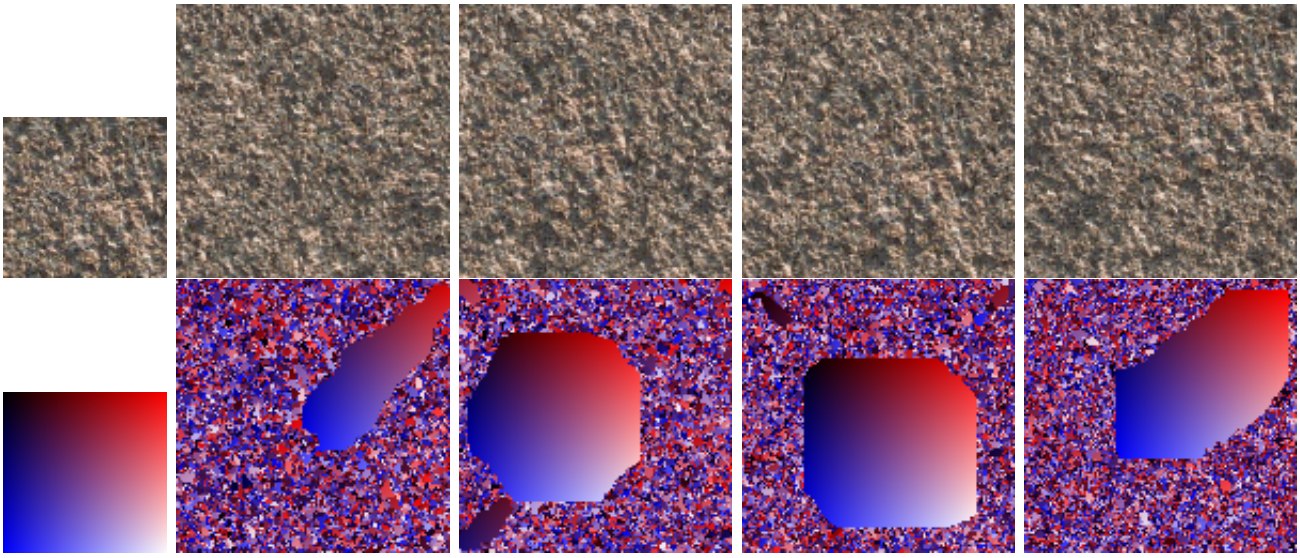


Figure 10: **Tolerance parameter variation. First row (from left to right):** sample image, results obtained with ε taking values 0.05, 0.1, 0.3, and 0.5. Patch size set to 5. **Second row:** Corresponding synthesis maps. The results are good for all the tested ε values. Results are still good for large values (0.3, 0.5), in contrast to the previous examples. This is mostly due to the noisy nature of the texture.

erated version are mostly indistinguishable from those obtained using the classical implementation. Both these results are visually very similar to the ones from the original paper.

For some particular color textures, where the color information is part of the texture, the accelerated version may produce incorrect results. This is caused by the dimension reduction performed to reduce the three RGB components to an unique luminance component. Most color textures can be correctly represented in gray levels (as all the color textures shown in this article so far) and are therefore correctly synthesized by the accelerated version. In some rare cases, the color content of images is not correctly captured by the first component of the PCA, so that the accelerated version yields artifacts. Such a case is displayed in figure 13.

7 Conclusions

In the present study, we have conducted a detailed analysis of the texture synthesis algorithm from Efros and Leung [8]. Extensive numerical experiments have been conducted. The main conclusion is that, except for some small scale textures, it is often hardly possible to set parameters in order to avoid both garbage growing and verbatim copy. Actually, we claim that most methods deriving from this original algorithm will synthesize textures by stitching together pieces from an exemplar. It therefore makes sense, since it is more efficient, to perform copy-pasting explicitly [18, 12].

We also provide a parallelized ANSI C code for both the classical and the accelerated version of the algorithm. Last, the accompanying online demo enables the user to synthesize images in reasonable time.

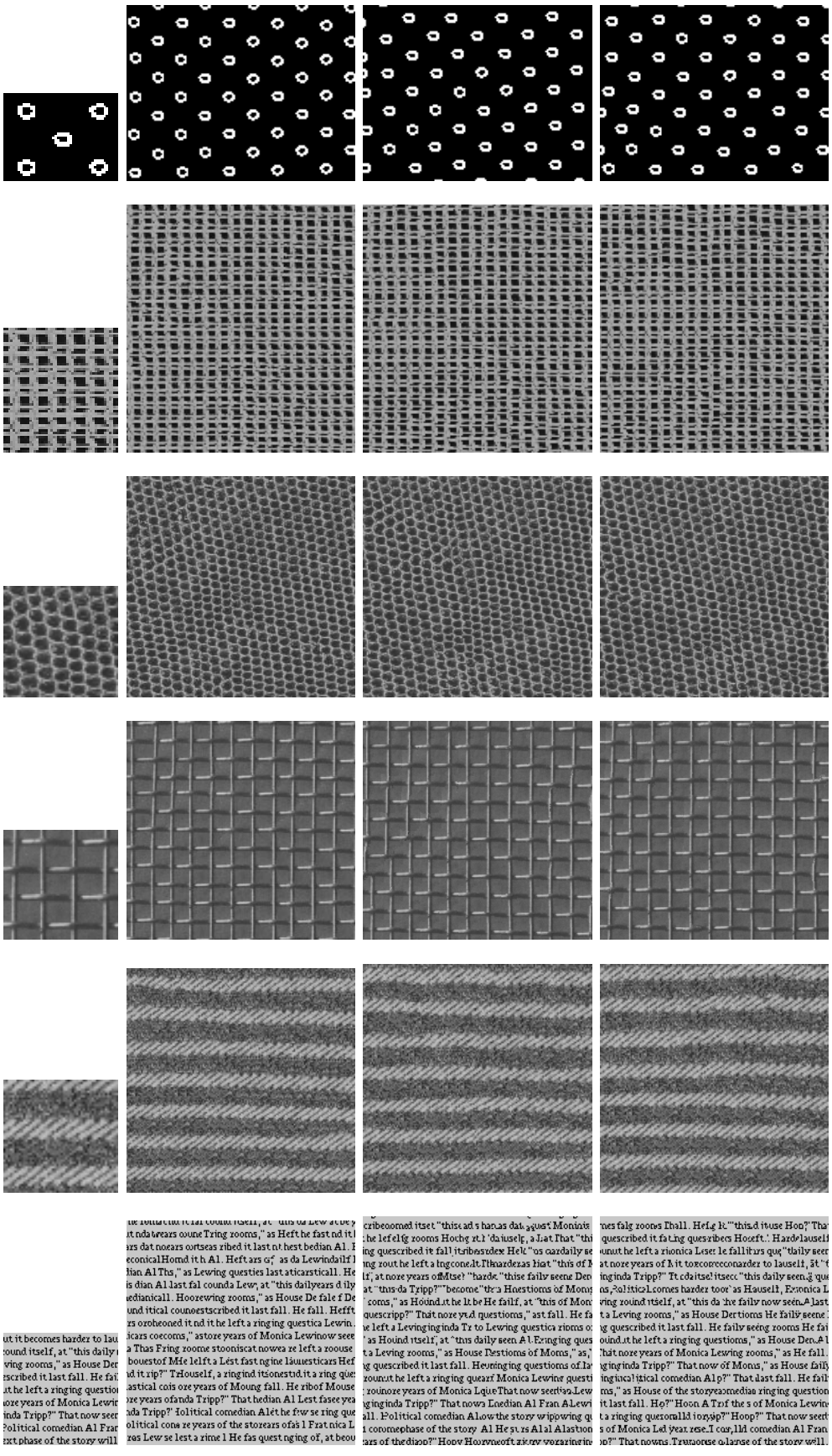


Figure 11: **Reproducibility of results.** Each row presents synthesis results on a texture from [8]. **From left to right:** sample image, result by Efros and Leung taken from [8], result obtained using our implementation of the method, result obtained using our accelerated version. Each time, all three results are visually very similar.

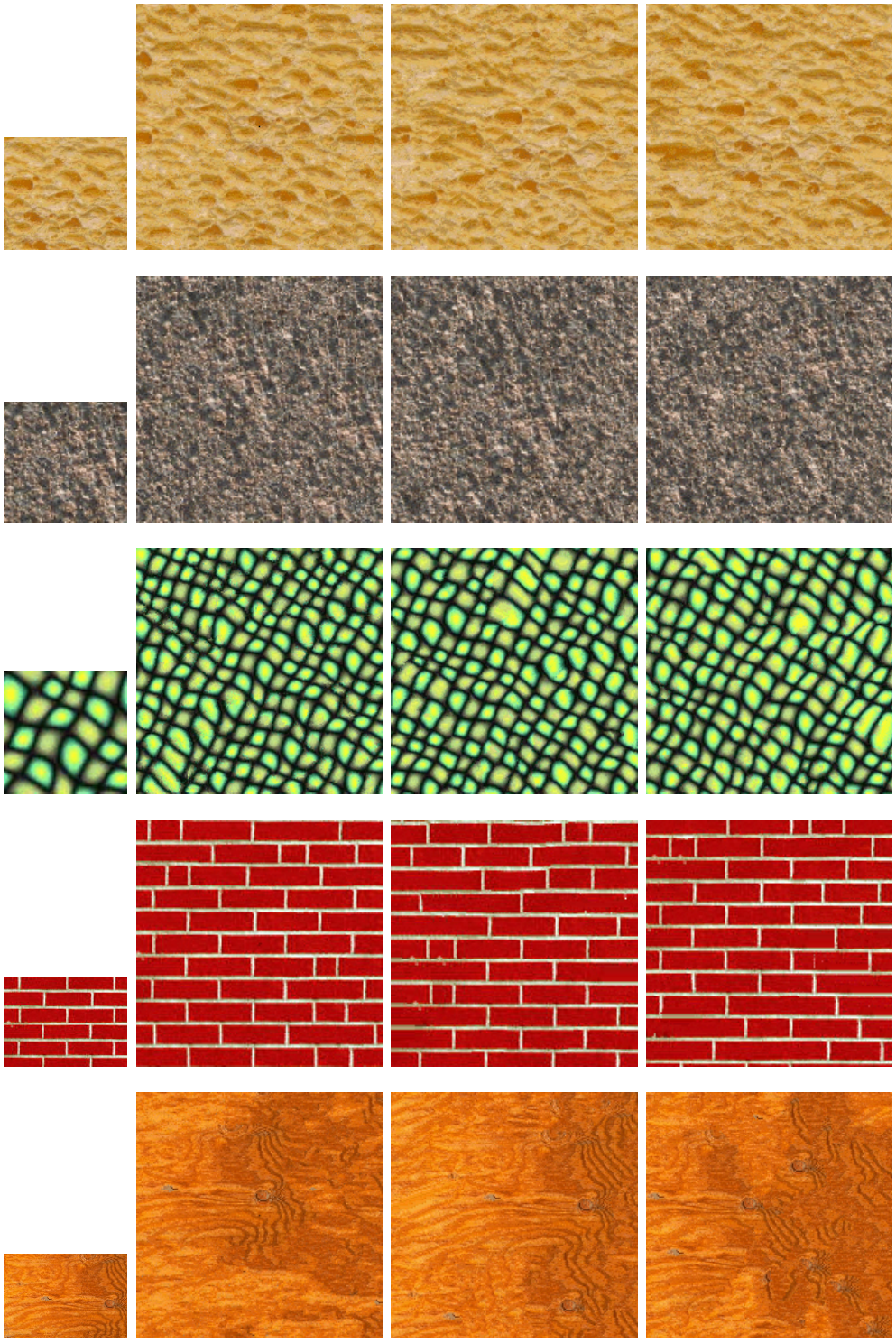
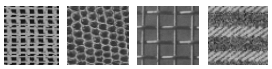


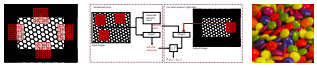
Figure 12: **Reproducibility of results.** Each row presents synthesis results on a texture from [8]. **From left to right:** sample image, result by Efros and Leung taken from [8], result obtained using our implementation of the method, result obtained using our accelerated version. Each time, all three results are visually very similar.

Figure 13: **Reproducibility of results. From left to right:** sample image, gray level version, result obtained using the classical implementation of [8], result obtained using the accelerated implementation. Some candies having different colors in the sample image are merged in the gray level version corresponding to the first component of the PCA. Color artifacts appear when using the accelerated implementation. This case is quite uncommon.

Image Credits

 by Efros and Leung [7].

 from Brodatz collection [2].

 by C. Aguerrebere, CC BY.

References

[1] T. Briand, J. Vacher, B. Galerne, and J. Rabin. The Heeger and Bergen pyramid based texture synthesis algorithm. *Image Processing Online (under review)*, 2013.

[2] P. Brodatz. Textures. New York, 1966.

[3] A. Buades, B. Coll, and J. M. Morel. A Review of Image Denoising Algorithms, with a New One. *Multiscale Modeling & Simulation*, 4(2):490–530, 2005. http://dx.doi.org/10.1137/040616024.

[4] G.R. Cross and A.K. Jain. Markov random field texture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(1):25–39, January 1983. http://dx.doi.org/10.1109/TPAMI.1983.4767341.

[5] J.S. De Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images. *Computer Graphics*, 31(Annual Conference Series):361–368, 1997. http://dx.doi.org/10.1145/258734.258882.

[6] 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 '01, pages 341–346, New York, NY, USA, 2001. ACM. http://dx.doi.org/10.1145/383259.383296.

[7] A. Efros and T. Leung. Texture synthesis by non-parametric sampling. http://graphics.cs.cmu.edu/people/efros/research/EfrosLeung.html. Accessed: 23/08/2013.

[8] A. Efros and T. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision Volume 2*, ICCV '99, pages 1033–1068, Washington, DC, USA, 1999. IEEE Computer Society. http://dx.doi.org/10.1109/ICCV.1999.790383.

[9] B. Galerne, Y. Gousseau, and J.M. Morel. Random phase textures: Theory and synthesis. *IEEE Transactions in Image Processing*, 20:257–267, 2010. http://dx.doi.org/10.1109/TIP.2010.2052822.

[10] B. Galerne, Y. Gousseau, and J.M. Morel. Micro-Texture Synthesis by Phase Randomization. *Image Processing On Line*, 2011, 2011. http://dx.doi.org/10.5201/ipol.2011.ggm_rpn.

[11] 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 '95, pages 229–238, New York, NY, USA, 1995. ACM. http://doi.acm.org/10.1145/218380.218446.

[12] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: image and video synthesis using graph cuts. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 277–286, New York, NY, USA, 2003. ACM. http://doi.acm.org/10.1145/1201775.882264.

[13] A. Lasram and S. Lefebvre. Parallel patch-based texture synthesis. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12, pages 115–124, Aire-la-Ville, Switzerland, 2012. Eurographics Association. http://dx.doi.org/10.2312/EGGH/HPG12/115-124.

[14] L. Liang, C. Liu, Y. Xu, B. Guo, and H. Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, July 2001. http://doi.acm.org/10.1145/501786.501787.

[15] E.P. Simoncelli and J. Portilla. Texture characterization via joint statistics of wavelet coefficient magnitudes. In *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, volume 1, pages 62–66 vol.1, 1998. http://dx.doi.org/10.1109/ICIP.1998.723417.

[16] L. 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.

[17] L. 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 '00, pages 479–488, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. http://dx.doi.org/10.1145/344779.345009.

[18] Y. Xu, B. Guo, and H. Shum. Chaos Mosaic: Fast and Memory Efficient Texture Synthesis. Technical report, Microsoft Research, April 2000.