Implementation of a Visual Model-Based Perceptual Image Hash for Content Authentication

Elena Canali

Partners: Aramanda Ottaviano Quintavalle
Samuele Andreoli

Abstract

Perceptual image hashes is a way to summarize the perceptual content of images. The goal of this report is to present our MATLAB®code, that computes perceptual image hashes following mainly method proposed in [5]. We use them to verify authenticity of images and to detect possible tampered region. In the first section of this report the perceptual image hash will be defined and briefly discussed; after that, will be present the state of the art. The third section is devoted to the presentation of our code in Matlab and to the discussion of our implementation choices. A short paragraph collect information about the database we used to test our code, and then, in section five, experimental results are discussed.

1 Perceptual Image Hash

Perceptual image hash is a summary of the perceptual content of an image. It has been widely investigated for authentication purposes: two images with the same perceptual content are required to have the same hash. On the other hand, if an image is tampered in a malicious way, its hash is required to be different from the original.

Properties that perceptual image hashes must have are not well defined: we referred to the properties listed in [2]. In order the hash to be useful, *compactness* is crucial: the size of the hash is required to be significantly smaller than the size of the image. Other needed properties are *collision resistance*, i.e. the probability that two different images have same hash has to be negligible and *preimage resistance*, i.e. the knowledge of the hash can not help to get the corresponding image and neither information about it. For perceptual point of view, *visual fragility* and *perceptual robustness* are crucial: the first property expects that the hashes of two slightly different images differ a lot; the second ensure that images with same perceptual content have same or very similar hashes.

Our implementation follows mainly the method proposed in [5]. Perceptual aspect is handled in features extraction process that is based on Watson's visual model (see for example [6]): it attempts to account for frequency sensitivity, luminance and contrast masking in the DCT domain in accordance with human's visual perceptual characteristics.

Our goal is to produce a perceptual image hash that is robust to rotation and scaling, and that is able to take over malicious tampered areas.

2 State of the Art

Image hashes are known and used since the last century but the perceptual aspect is of recent concern. Although already exist works that refer to this issue, [5] is the first that really focus on human's visual perceptual characteristics. Moreover none of previous works were really able to detect changed image regions.

Previous algorithm that compute perceptual image hashes were divided in two main categories: image-block-based methods and feature-point-based methods. The former are able to detect image content changes but the detection accuracy is not reliable; furthermore they are unable to handle geometric distortions. The latter can be used to deal with geometric transformed images but, because feature point are sparse, is not suitable to process with texture areas; moreover detection accuracy is still not good. We combined image-block-based features and key-pointed-based features to generate a more robust code, sensitive to content change.

3 Implementation

The following section will contains the description of our code's functions. We will focus in particular on the difference between the scheme proposed in [5] and our choices. How the parameter values are assigned is not concern of this section: we will discuss about this issue in the following section.

3.1 Extract Features

The function extract_features take as input:

- **image_0**: a matrix of single precision real numbers, representing the image we want to operate with;
- **s1, s2**: the projection rates for compression matrices;
- **tc**: a string containing information about which kind of compression to use. The possible values for this parameter is 'Gauss' or 'Bernoulli';

- **f_tform**: string indicating the transform used during computation. The possible values are 'SIFT' or 'SURF'.

The outputs are:

- **F_0**: 1x5 cell containing intermediate hash concerning *image_0*;
- **Gs1, Gs2**: compression matrices applied to the pre-digest.

The implementation of this function follows mainly the scheme suggested in [5]. First geometric invariants are extracted from*image_0*. We introduced the possibility to choose between two different transform: the extraction process is performed according to the value of *f_tform*. If *f_tform*='SIFT', frames and descriptors are extracted using *vl_sift* [1]; to obtain sparse representation of the image, we compute a single-level one-dimensional wavelet decomposition.
We proposed an alternative transform to SIFT: if *f_tform*='SURF', features and respective locations are detected and extracted using *detectSURFFeatures* and *extractFeatures*. Also in this case, we apply afterwards the 1-level db1 wavelet transformation.

Both transform are able to extract features that are invariant for geometric transformation such as rotations or translations. The main difference is the number of features extracted and the size of each one of them: when SIFT is used, the amount of extracted elements in much higher and each feature is a 128 vector; SUFT produce instead less vector that have 64 elements each. For this reasons, SIFT transform produces more robust results, in particular with heavy geometric distorted images, SURF gives faster outcomes still keeping good results.

After this first step,extraction of block-based features is performed. We split the image into 8x8 non-overlapping blocks and we apply the DCT transform relying on the Watson's DCT-based visual model. The resultant weighted DCT matrix is then reshaped using the zigzag function[2].

Lastly we compute the output's compression matrices: also in this case the type of the compression matrices depends on the value of an input's parameter. If *tc*='Gauss', two Gaussian random matrices are generated, according to the value of *s1* and *s2*. We introduced the possibility to get Bernoulli matrices, setting *tc*='Bernoulli': we built this matrices using the Matlab function *randi[0,1]* getting matrices of approximately half 0's and half 1's. Matrices are then modified in order to fit Bernoulli distribution. The compression operation consists in the multiplication with the matrix just compute.

[1] See the Matlab library [4].
[2] See [1]

3.2 Encrypt

The function *encrypt* take as input:

- **F**: the unencrypted digest (output of *extract_features*);
- **K**: secret seed.

Output of the function is:

- **F_E**: the hash.

This function follows the operations shown in [5]. First keys are generated using K and saved in an array k. Afterwards the encryption algorithm is applied to F using the keys stored in k to get the output F_E.

3.3 Verifier and decrypt

The function *verifier* is responsible to compare original and tested images. The inputs of the function are:

- **image_t**: a matrix of single precision real numbers, representing the image we want to compare to the original one;
- **F_E**: the hash of the original image;
- **K**: the seed used in the encryption process to produce the hash of the original image: we will use the same seed to produce the hash of *image_t*;
- **Gs1, Gs2**: compression matrices;
- **Tr_a**: threshold used to decide whether some geometric distortion occurred or not;
- **f_tform**: string indicating the transform to use in features extraction process. Can be either 'SIFT' or 'SURF'.

The outputs of *verifier* are:

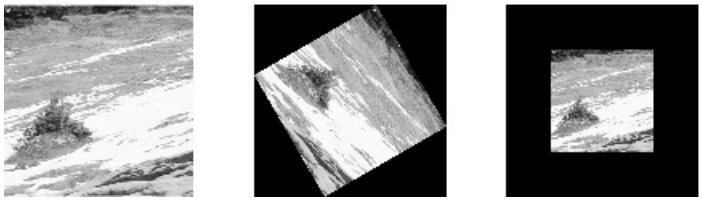
- **d**: Euclidean distance between original image's blocks and the correspondent blocks of *image_t*;
- **d_a**: Euclidean distance between each pair of matched feature points;
- **blocks_RGB**: blocks of the image to be verified, stored in three levels.

First F_E is decrypted using the function *decrypt* that take as input F_E and K and performs the inversion of the function *encrypt*. The output of this function is F_D i.e. the unencrypted hash.

Afterwards features of *image_t* are extracted using the same operations as in *extract_features*. Most similar features and related positions are then extracted, using matching functions that depends on the value of *f_tform*.

The Euclidean distance *d_a* between matching features is then computed and used to determine whether a geometric transformation occurred or not: for this purpose we compared the minimum of *d_a* with *Tr_a*. Unlike what suggested in [5], we perform this operations at this point to avoid distortion due to numerical imprecision. Moreover other equivocal situation are avoided: when images are tampered pasting some region coming from the same images, matching features can have some ambiguity because we can have an huge number of false matching. This does not happend here, because we check the presence of geometric distortion before working with affine transform.

If some geometric transformation is detected the straightening process of the image under review is performed. First the parameters of the transformation are calculated comparing matching points. We used the Expectation-Maximization algorithm [3] to solve this linear system because, unlike the Matlab function *linsolve*, this allows to exclude strong outliers, i.e. false matching features detected in the matching algorithm. In the Expectation–Maximization, the third column of the matrix representing the affine relation is forced to be $(0, 0, 1)^T$ because Matlab expect this kind of values in a affine transform matrix. *image_t* is then substituted with its straightened, cropped and resized version: the last two operations allowed to remove black edges due to geometric transformation. Indeed if this operations are not performed, the image we obtain is not equivalent to the original one but is its smaller version: the following picture shows what would happen if we removed cropping and resizing.



Afterwards we apply the DCT transform to 8x8 blocks of *image_t* using Watson's frequency sensitivity matrix and then compress the resultant signal using *Gs2*, similarly as in *extract_features*. Lastly we computed the Euclidean distance *d* between the blocks of the original image and the tested one.

The function *verifier* deals with the decision to determine whether the tested image is related to the original one through some geometric transformation; the presence of possible tampered regions is decided in *main* and localized in *localization* that will be discussed in the next sections.

[3] see [3] for a description of the algorithm

3.4 Localization

The function *localization* is used only when images are declared inauthentic. The inputs of the function are:

- **d**: Euclidean distance between original image's blocks and the correspondent blocks of image to be verified;
- **d_a**: Euclidean distance between each pair of matched feature points;
- **blocks_RGB**: blocks of the image to be verified, stored in three levels;
- **Tr_a**: threshold used to decide whether some geometric distortion occurred or not;
- **Tr_1**: threshold to be used in the localization of tampered regions, when no geometric distortion is found;
- **Tr_2**: threshold to be used in the localization of tampered regions, when any geometric distortion is found.

The output is:

- **blocks_RGB**: blocks of the to be verified image, where blocks corresponding to tampered regions are highlighted with color red;

As already mentioned, this function is used when any kind of tampering is found: this is decided in *main* comparing the maximum value of d and Tr. First the check of geometric transformation is performed (in the same way as it is computed in *verifier*): if geometric transformation is found, every block which corresponding value of d is bigger that Tr_2, is marked color red; if not, the same operation is done over blocks which corresponding value of d is bigger then Tr_1.

3.5 Main

The *main* function collects all the steps to compute the perceptual hash of images and to perform the authentication process between two pictures in input. The inputs are:

- **img1**: the first image;
- **img2**: the second image;
- **tc**: string containing information about which kind of compression to use. The possible values for this parameter is 'Gauss' or 'Bernoulli';
- **f_tform**: string indicating the transform used during computation. The possible values are 'SIFT' or 'SURF'.

The outputs of *main* are:

- **authenticity**: array of two positions that can be 0 or 1: the first position is 0 if tampered region is detected, 1 otherwise; the second position indicate presence or absence of geometric transformation in the same way;
- **blocks_RGB**: blocks of image to be verified where blocks corresponding to tampered regions are highlighted.

The function *main* calls the functions we already discussed to state if the two input's images are perceptually equivalent or not.

First optional inputs are managed: if the number of *main* input arguments is less then 3 or 4, i.e if the value of the parameters *tc* and *f_tform* is not specified, default value are assigned (*tc*='Gauss', *f_tform*='SIFT').
Afterwards the values of the thresholds and projection rates is set up: *Tr*, *Tr_a*, *Tr_2* and *Tr_1* are the thresholds used respectively to decide the authenticity of the images, to determine if some geometric transformation happened, to localize distortions when geometric transformation is detected and when not; *s1* and *s2* are the projection rates. The value of this parameters is assigned thanks to the tests that we will discuss in the next section. The user is supposed to change them if wants to test the algorithm with different values.
The function *extract_features* is then applied to the first image (considered as the original image with which compare the second image) and the resultant intermediate hash is encrypted with *encrypt*. Afterwards *verifier* is called and, thanks to its outputs, it is decided whether the second image a tampered version of the first or not. The minimum of the values stored in *d_a* is compared to *Tr_a*: if the minimum exceeds the threshold, some geometric transformation is detected. The maximum of *d* is compared to *Tr* in order to determine if some tampering happend or not.

3.6 Perform algorithm

perform_algorithm is a simple script that show the result of the authentication algorithm performed in main: we preferred to build a separate script that deals with this operation because in this way the structure of *main* can be easily adapted to perform multiple tests.

The user is supposed to change the value of *path* at the beginning so that the folder containing images can be reached. When *perform_algorithm* is ran, the indicated folder is opened and the user has to select the first image and afterwards a similar operations is demanded for the second image. At this point main is call and the response of the computation is printed: if the image declared geometrically transformed, "Image_t has NOT undergone geometric manipulation." is printed and the image is shown, otherwise "Image_t HAS undergone geometric manipulation." is printed and the straightened picture is shown. If the image is declared tampered, "Image_t IS tampered." is printed, otherwise "Image_t is NOT tampered."; furthermore the tampered image with non authentic regions colored with red is shown.

4 Data Base

We collected 50 grayscale .TIF images of dimension 512x512 to test our implementation. We modified 41 of those images, adding some tampered regions and change 23 pictures applying different kind of geometric transformation; we also applied at some other images other kind of noise, (Gaussian noise) using suitable Matlab functions. For images that are altered only in some specific region (the 41 images we mentioned before), we built also black and white maps that highlight tampered regions: white corresponds to modified areas, black indicates unchanged regions.

We chose different kind of pictures with different perceptual characteristic: we altered them trying to cover as many cases as possible so that to get as much reliable test result as possible.

We used pairs of images original images-modified images to test our code; in tests where we analyzed the value of the parameters *s2*, *Tr_1* and *Tr_2* we also used the maps that trace tampered regions in order to be able to distinguish true detected tampered region and false one.

5 Test and Experimental Results

In this section we will discuss how we tested our implementation and we will present experimental results. Test are performed with a computer with a Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz 2.60GHz, 4,00GB RAM.

5.1 Test of parameters

We wrote some Matlab scripts that use the already introduced functions or their slightly adapted version. We ran our code changing value of one parameter and keeping all the other fixed. This allowed us to establish the best value of each parameter so as to achieve best performances.
We performed tests on geometric transformed pictures or on images with tampered regions, depending on the parameter under review. Each of the following subsection will contain a brief argument about different test we performed.

5.1.1 Test of parameter *Tr_a*

The parameter *Tr_a* represents the threshold that discriminates the presence or the absence of geometric transformation. To test the value of this parameter, we wrote a suitable script and ran it over our database of geometric transformed images.
The value of *Tr_a* is compared to the minimum value of *d_a*: we modified the function *verifier* and consequently the function *main* in order to output 1 when a geometric transformation is detected and 0 if not. This helped us to establish the rate of images declared geometrically tampered.

We first set the range of values we wanted to test: we start with *Tr_a*=0 and we end with *Tr_a*=3 by step 0.1. For each value of *Tr_a* we called the function *main* over all the 23 couple of original-geometric transformed images; moreover we applied some Gaussian noise to the pictures, so that the same perceptual content of the original images is kept and we computed the same operations for pairs of original-with Gaussian noise images. This operations allowed to perform a threshold standard test: we are interested to identify true positives (if p is an event, $\mathbb{P}(p \text{ is detected} \mid p \text{ is true})$ and false positives (if p is an event, $\mathbb{P}(p$ is detected $\mid p$ is false). At this purpose we stored how many images are declared geometrically tampered in geometric transformed and Gaussian corrupted case: in this way we gathered information about true positive and false positive.
At the end we collect all the information and shown them in a graph: on the horizontal axis the value of *Tr_a* is indicated while the vertical axis show the rate of geometric transformed images detected.

We performed this test for all the possible values of *f_tform* and *tc* but the resultant graphs are similar in all cases: we claim that *Tr_a*= 0.5 gives best results for all the possible combination of parameters. We report in the figure below graph representing false positive detected for *tc*='Gauss' and *f_tform*='SIFT'. We also show graphs of true positive detected for both kind of transform.

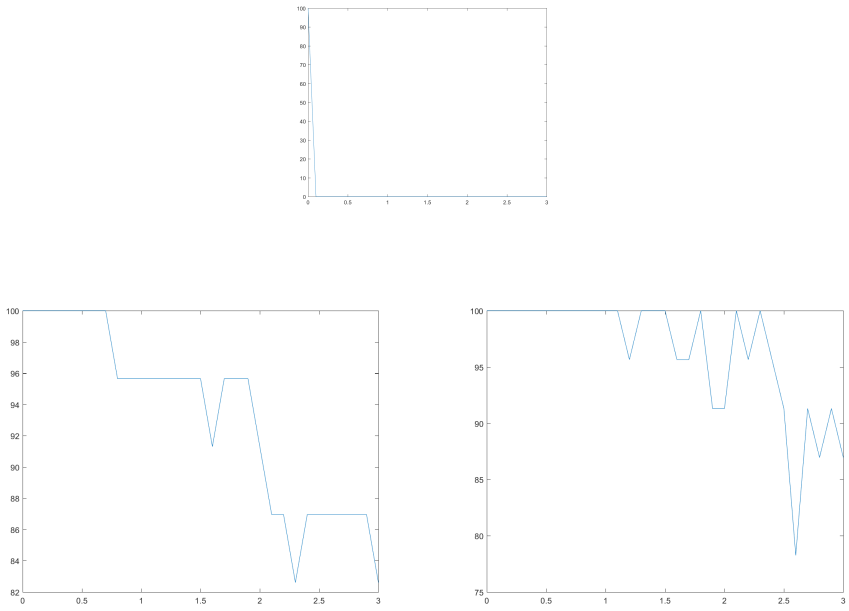


Figure 1: On the top false positive rate for *f_tform*='SIFT', *tc*='Gauss'. On the bottom we have on the left true positive rate for *f_tform*='SIFT', *tc*='Gauss', on the right true positive rate for same *tc* and *f_tform*='SURF'.

5.1.2 Test of parameter *s1*

The parameter *s1* represents the projection rate used to build *Gs1*. Also for this test we referred to the database of geometrically transformed images while *Gs1* is crucial in features matching: achieving good results with this images leads to get good results in every situation.

Similarly of what was done for *Tr_a*, we tried to execute *main* for different values of *s1*: we start with *s1*=5 and we end with *s1*=10 by step 0.5. The functions *verifier* and main are modified so to output similarity between the original images and the straighten versions coming from the matching algorithm. For each value of *s1* this similarity is computed for all the images, and stored in an array; finally a graph shows the average of the similarities for each value of *s1*. We accepted the value of *s1* that minimize the similarity: this value allows a better estimation in affine transformation.

Also in this test, we tried to perform the algorithm for different values of *f_tform* and *tc*.
For *f_tform*='SIFT' we obtained the following graphs:

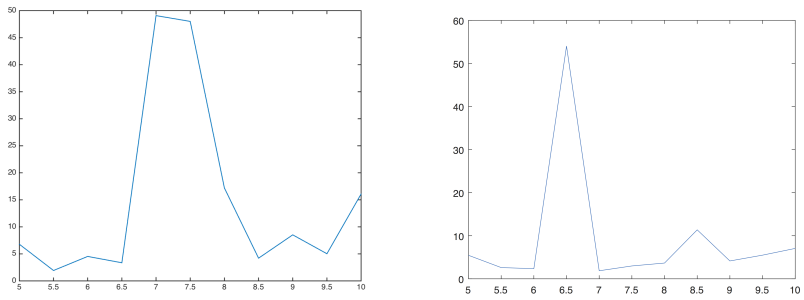


Figure 2: *f_tform* is set equal SIFT. On the left Gaussian compression is used; on the right Beronoulli compression is used. X-axis shows the value of *s1*

In the Gaussian case, good results are obtained for $s1 \in [5.5, 6.5]$ while for Bernoulli case, best outcomes are given by the interval $[5, 6]$. We decided to set $s1 = 6.3$ as suggested in [5] as this value do not produce bad results in Bernoulli case and ensure on the other hand good results in Gaussian case.
According to graphs below, *f_tform*='SUFR' outcomes worst performances: corresponding value of *s1* produces higher similarities. However the value 9.5 gives results comparable to the previous case.
In both cases our graphs show some abnormal peaks due to the estimation of non-invertible matrices in the computation.

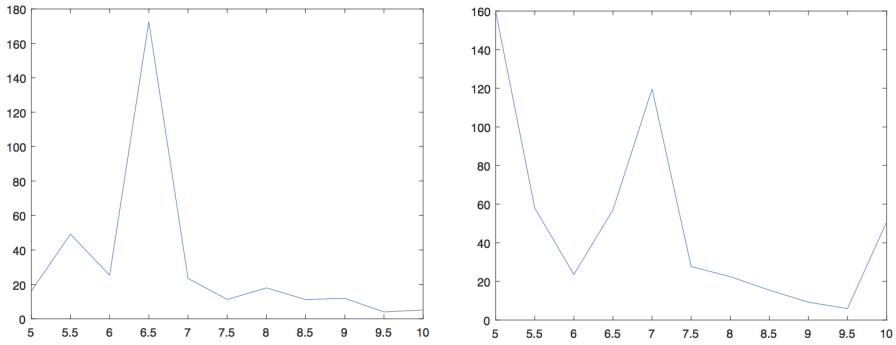


Figure 3: *f_tform* is set equal SURF. On the left Gaussian compression is used; on the right Beronoulli compression is used. X-axis shows the value of *s1*.

5.1.3 Test of parameter *s2*

From now on, we will consider for our tests the database containing images with tampered regions but not undergone to geometrical transformations. We exclude from our database such images because the parameters under review are used after the straightening process: working with geometrically transformed pictures would have meant to introduce possible noise and therefore to compromise our results. For this kind of tests, we deleted from the function *verifier* the code that handle the geometric transformation, that is useless in this case.

The parameter *s2* represents the projection rate used to build *Gs2*. The compression matrix *Gs2* is used to compress image blocks that are involved in the computation of d: an alteration of the value of *s2* produces variations in d and therefore compromise the authenticity statement and the detection of tampering regions. To test this parameter we exploited the black and white maps built to trace modified regions: they allowed to determine if the localization of the tampered areas happened correctly or not and to calculate the rate of detected tampered regions. To perform this operations we built a specific function named *compare_maps* that takes as input the original map, *map_or*, and the map of detected tampering regions, *bwmap* and outputs the rates of real detected tampering, *detected_real_tampering*, and of false detected tampering, *detected_false_tampering*. In order to do that, each 8x8 white block of *bwmap* is compared to the correspondent block of *map_or*: if this block is more than half white, we increased the value of *detected_real_tampering*, otherwise the one of *detected_false_tampering*.

The function *main* is modified so to take as input also the map of the tampering regions. Similarly as the *main* used in test for *Tr_a*, we corrupted images with Gaussian noise to have information about false positive. Like in the previous cases, it is output 1 when the image is declared authentic, 0 otherwise; we also added to the outputs, the value returned to the function *compare_maps*. We

changed also the function *localization*, that in this test output the map of the detected tampered region: blocks that corresponds to non authentic areas are colored with white, while the others are black.

We choose range of value for *s2*, from 5 to 10 by step 0.5, and we called the function *main* for all the possible images; for every tampered image and for every picture corrupted with Gaussian noise that are declared inauthentic we increased respectively the value *ND* or *NFD* (that are the variables that count the number of true and false positives).
At the end of the computation, we created graphs representing the rate of true positives and false positives. Afterwards we checked also the false/true positive blockwise for all the values of *s2* we analyzed. Comparing true and false positive, we claim that the value for *s2* that ensure best performances is to be chose the range $[5, 9]$.
The following graphs shows the blockwise true and false positive. The shown results allow to set the value of*s2* to 9, as suggested in [5].

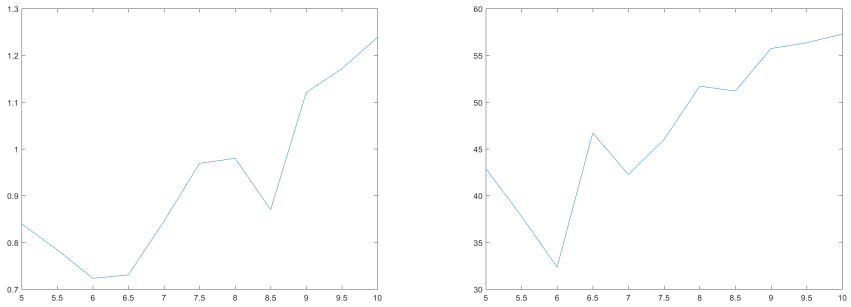
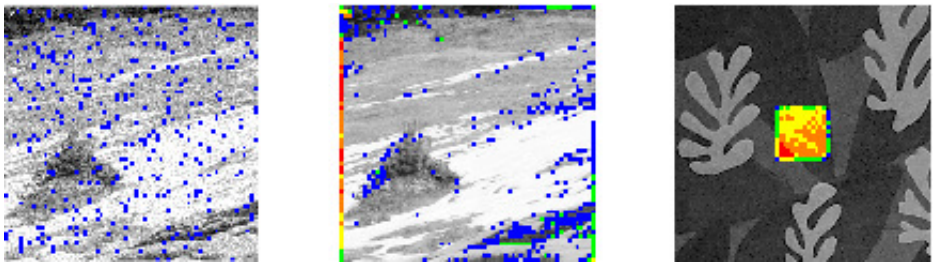


Figure 4: On the left is shown the blockwise false positive rate; on the right is shown the blockwise true positive rate; x-axis: s2

5.1.4 Test of the parameter *Tr*

For the following tests, we needed to restrict the range of values to try. We create a scale that allowed to highlight different behavior of the similarities between the block-based features. We applied this scale to different kind of pictures (corrupted with AWGN, rotated by 10 degrees, tampered with foreign region) that corresponds to different type of images we worked with. Values lower than 1 are left unmarked, the other are marked with a color from blue (1-2) to red (¿ 5). Thanks to this set of outcomes we decided the following testing range: $Tr \in [0, 4]$, $Tr\_1 \in [2, 7]$, $Tr\_2 \in [4, 9]$.



The parameter Tr is used to determine the authenticity of the images. Its value is compared to the maximum of d. It has to be such that noise that do not affect the perceptual content of the image is tolerated while malicious tampering is not.
The script used to test this parameter is very similar to the one for Tr_a: we applied the authentication algorithm twice, once to images with tampered regions and once to images corrupted with Gaussian noise. The function *main* is modified in order to output 0 when authenticity is declared, 1 otherwise. We started with Tr=0 and and with Tr=4 by step 0.5: for each of this values of Tr we stored the number of tampered images (true positive) and the number of pictures corrupted with Gaussian noise (false positive) declared inauthentic.
We used the Euclidean distance in the computation but we left the option to change it if needed.
The following graphs shows the outcomes of this test: false positive are 0 for $Tr \geq 2.5$ and true positive are 100 percent for $Tr \leq 2.5$. However we chose to set Tr=3 because we preferred to loose some true detection rather than to risk some false detection.

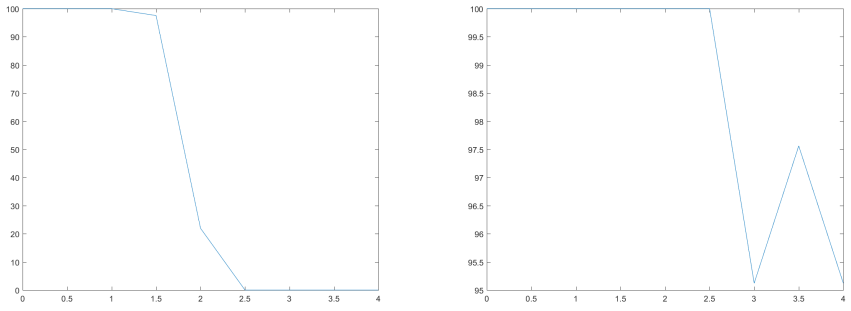


Figure 5: Left: false positive rate, right: true positive rate; x-axis: Tr

5.1.5 Test of the parameter *Tr_1* and *Tr_2*

The parameters *Tr_1* and *Tr_2* are used in the function *localization* to locate tampered regions respectively when no geometric transformation is detected and when some geometric transformation is found. Also to perform the test of this parameter, we needed to slightly change function *main*: the maps of tampered regions is added to the inputs and the output of *compare_maps* is returned. *main* call the function *localization* that is the same used for the test of *s2* with different threshold given in input (respectively *Tr_1* and *Tr_2*): this function will output a black and white map highlighting the detected tampered areas: this map is then compared to the original map thanks to *compare_maps*.
As in the other cases, we perform the testing algorithm for all the images of our database, for different values of the parameters: we started with *Tr_1*=2 and *Tr_2*=4 and we ended with *Tr_1*=7 and *Tr_2*=9, both by step 0.5. For either test of *Tr_1* and of *Tr_2*, graphs show rates of true positive and false positive detection for each value of the parameters. The results shown in graphs are similar: we show in the following figure outcomes of the *Tr_1* test. A good tradeoff between true and false positive is given by the value 3: in the same way we set *Tr_2*=5.

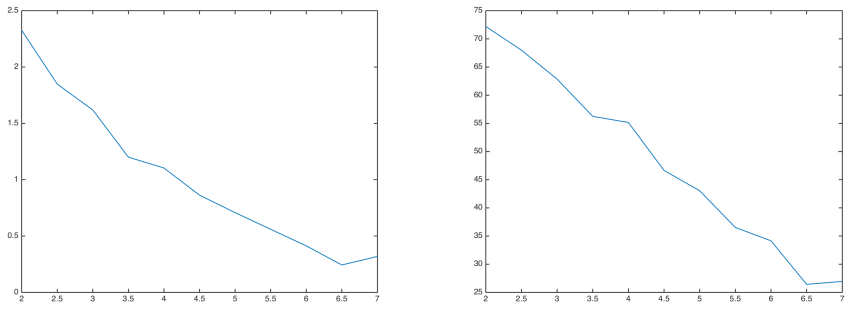


Figure 6: On the left, blockwise false positive rate. On the right, blockwise true positive rate. X-axis represents the value of *Tr_1*.

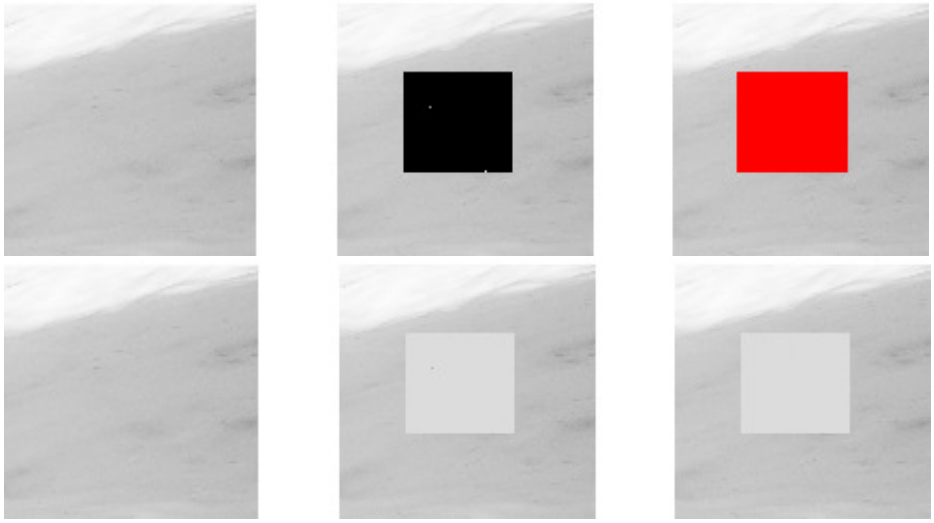
with this values we obtained about 50% of true positives: this is still a good result since it considers also images where tampered regions have similar perceptual content to the original one and therefore have not to be marked.

5.2 Test of the algorithm

According to the tests we already discussed, we identified values of each parameters that allowed to achieve best results. We set parameters in *perform_algorithm* with this values. This section is devoted to present and discuss performances of the original code, i.e. results of *perform_algorithm*. We will collect sets of three images: the first is the original one, the second is tampered and the third is the resultant picture of *perform_algorithm*, performed with the former images.
We start with a very good outcome:

tampered region is 100% detected, with high precision.
Our purpose was to detect and localize perceptual different changes in the images. The following picture shows a tampering where the compromised region is of two type: some areas keep the same perceptual content (flowers pasted over other flowers), some others are drastically changed (flowers pasted over leaves). Our algorithm worked well also in this picture: perceptually equivalent areas are not marked, while different ones are colored with red.

Although this is a very good outcomes, this compromised true positives rates in our *Tr_1* and *Tr_2* tests: this tests declared non detected areas as missed and do not concern about perceptual content. The results we obtained in this tests must be interpreted as including also this kind of situation: for this reason, outcomes like 55% or 45% of true positives (at the value chosen for *Tr_1* and *Tr_2*, respectively corresponds this rates of true positives) are not to be considered as bad.
We now present a slightly different case: we compare two situation where the tampering is the same but the perceptual effect is deeply different.



The original image is the same; the tampered area is a black centered square in the first case and a white centered square in the second. Since background of the original image is light, the black square is perceptually visible while the white square is much more difficult to see. Our algorithm detects completely the tampered region in the first case and is not able to detect it in the second: this outcomes can be considered sill quite good.
The following set of images summarizes the already discussed perceptual issue.

Spots that have similar color or luminance to the background are not detected whereas perceptual differences are marked (white and black spots are detected

better the gray ones).

As already mentioned, another good result we achieved, is robustness against heavy copy-move forgeries. The following pictures prove the correctness of the detection in this situations.

Lastly we show some interesting results obtained with geometrically transformed images. The original picture is rotated by 22 degrees. The straightening process worked very well is this picture and detection rate is quite good.

On the other hand, small imprecision in affine transform estimation and straightening process can produce high number of false positives and compromise the result: the following pictures present this kind of scenario.

6 Conclusion

The perceptual image hash we proposed, presents some little improvement compared to the one proposed in [5]. Greater improvements are achieved in authentication and tampering localization algorithm: geometric manipulation, tampering detection and localization gives satisfactory results.
Some imprecision are still present, expecially in affine transform estimation due to false matches in the extracted features. Future develop of the algorithm might focus on compression issue, that mainly cause this false matches.

References

[1] wyyang53@hanmail.net, 2012.

[2] Lv, X., and Wang, Z. J. Perceptual image hashing based on shape contexts and local feature points. *IEEE Transactions on Information Forensics and Security 7* (2012).

[3] Moon, T. K. The expectation-maximization algorithm. *IEEE Signal Processing Magazine 13* (1996).

[4] Vedaldi, A., and Fulkerson, B. The vlfeat team, http://www.vlfeat.org/index.html.

[5] Wang, X., Pang, K., Zhou, X., Li, L., and Xue, J. A visual model-based perceptual image hash for content authentication,. *IEEE Transactions on Information Forensics and Security 10* (2015).

[6] Watson, A. B. Dct quantization matrices visually optimized for individual images. *Proc. SPIE* (1993).