IM01: Automatic Copy-Move Detection

Corlou Elias & Palagi Théo

November 2025

1 Introduction

Digital image modification has become a common practice, facilitated by powerful editing tools. While these tools often serve artistic or corrective purposes (removing unwanted elements, reconstructing textures), they can also be used maliciously to falsify documents or visual information. One of the most common forgeries is "Copy-Move", which involves copying a part of an image and pasting it elsewhere in the same image to hide an object or clone a crowd.

In this project, we focused on the *PatchMatch* algorithm [1], initially designed for fast structural image editing (inpainting, reshuffling). Our objective was twofold: first, to understand and implement this dense correspondence algorithm, and then to repurpose it from its original function to use it as an analysis tool capable of automatically detecting duplicated areas.

The experience gained from previous projects allowed us to organize our work effectively. The main mistake we made in the past was trying to fully and theoretically analyze a research paper before writing a single line of code. This time, we adopted a more interesting approach to the paper. We started with a "diagonal" reading of the founding paper *PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing* [1], extracting only the key information needed to start: random initialization and the concept of propagation.

Once the basic building block was functional (reconstructing an image by itself), we made the model more complex step by step by following the paper. Finally, faced with the difficulty of moving from "correspondence" to "detection", we relied on a second specialized article, *Automatic Detection of Internal Copy-Move Forgeries in Images* [2], to implement a robust detection pipeline (error maps, masks).

This document details our approach. We first present the internal working of PatchMatch and our Python implementation. We then explain how we adapted the algorithm for forgery detection by introducing spatial constraints. Finally, we describe the post-processing pipeline necessary to obtain binary detection masks and discuss the obtained results.

We work with images, modeled as functions $\Omega \to \mathbb{R}^d$, where $\Omega \subset \mathbb{Z}^2$, $d = 1$ for grayscale images and $d = 3$ for color images, with d representing the number of channels.

2 The PatchMatch Algorithm

The core of our project relies on the PatchMatch algorithm. The fundamental problem it solves is the computation of a *Nearest-Neighbor Field* (NNF). It

therefore computes an NNF by exploiting the local similarity property ubiquitous in natural images. This means that if a patch a in A is matched to a patch b in B, then the neighbors of a are more likely to be matched to the corresponding neighbors of b. This idea is complemented by a random search for the best possible matching patches, then iterated a fixed number of times. For many applications, a few iterations (about 5) provide an acceptable result.

2.1 NNF Definition

Let there be two images A and B (which can be identical in the Copy-Move case). For each patch (square neighborhood of pixels) of coordinates (x, y) in image A, we look for the patch in image B that is most similar to it according to a distance D (usually the sum of squared differences, SSD). The NNF is defined by the offset function $f : A \mapsto \mathbb{R}^2$, such that:

$$f(x, y) = \arg\min_{(u,v)} D(\text{Patch}_A(x, y), \text{Patch}_B(u, v))$$

For a given patch a in A and its nearest neighbor b in B, the offset is then $b - a$.

2.1.1 Random Initialization

The algorithm starts by assigning a random correspondence from patches of A to patches of B. Even if a random assignment is unlikely to be correct for a given pixel, over the whole image, it is statistically probable that *some* pixels are correctly matched.

2.1.2 Propagation Step

This is the key step that exploits image coherence. The idea is as follows: if patch a centered on pixel (x, y) corresponds well to patch b centered on pixel (x', y') in the target image, then it is very likely that its right neighbor $(x+1, y)$ corresponds to the right neighbor of the target $(x' + 1, y')$.

In our `propag` function, we traverse the image and try to improve the current offset by looking at the neighbors' offsets.

- **Forward Pass:** We traverse from top to bottom, left to right. We test if the offset of the pixel above $(x, y - 1)$ or to the left $(x - 1, y)$ offers a better distance for the current pixel.
- **Backward Pass:** We traverse from bottom to top, right to left. We test the neighbors below $(x, y + 1)$ and to the right $(x + 1, y)$.

Formally, if $D(v)$ is the distance obtained with an offset v, the new offset $f(x, y)$ becomes:

$$f(x, y) = \arg\min_{v \in \{f(x,y), f(x-1,y), f(x,y-1)\}} D(v)$$

2.1.3 Random Search

Propagation allows good matches to "flow", but it can get stuck in local minima. Random search allows escaping by testing random offsets in a concentric neighborhood around the current best solution. Taking $v_0 = f(x, y)$ a given

offset, we try to improve the current offset $f(x, y)$ by testing a certain sequence of candidates around it, as follows:

$$u_i = v_0 + w\alpha^i R_i$$

With $R_i \sim \mathcal{U}([-1, 1] \times [-1, 1])$, w the maximum search radius and α a coefficient taken equal to $\frac{1}{2}$. We then examine each patch for $i = 1, 2, 3, ...$ until the window is 1 pixel. The search radius w decreases exponentially at each step ($w \cdot \alpha^i$ with $\alpha = 0.5$).

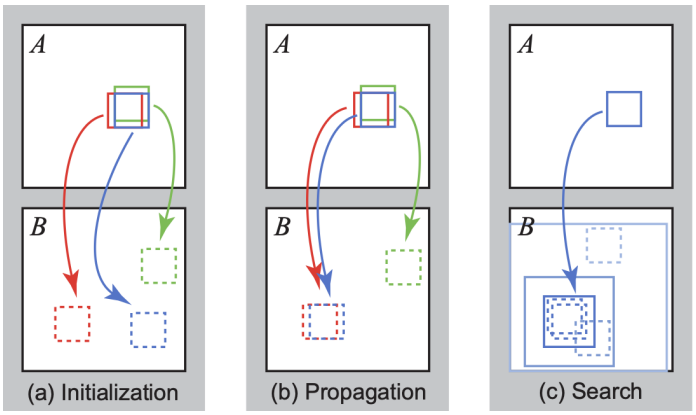


Figure 1: Phases of the nearest neighbor search algorithm.
(a) patches initially assigned randomly; (b) the blue patch checks its top (green) and left (red) neighbors to see if they can improve the blue mapping, propagating good matches; (c) the patch searches randomly for improvements in concentric neighborhoods.

2.2 Reconstruction (Remap)

To verify our basic implementation, we used the `remap` function. The goal is to reconstruct the original image from itself using the calculated offsets. If the algorithm works, the reconstructed image must be identical to the original (because each patch should find itself or a very similar patch). This works perfectly for two equal images. We also tried to reconstruct an image B from the correspondences of its patches in image A. We also obtain a perfect result, exactly what we had at the start.

Figure 2: Image A

Figure 3: Image B

Figure 4: Remap

3 Adaptation to Copy-Move Detection

Once PatchMatch was functional for reconstruction, we oriented the project towards "Copy-Move" detection.

3.1 The Self-Matching Problem

In a classic inpainting application (hole filling), we look for any similar patch. But to detect a copy in an image A towards A, the "naive" PatchMatch algorithm converges to the trivial solution: $f(x, y) = (0, 0)$ (each pixel corresponds to itself, zero distance).

To force the algorithm to find *the copy* and not the original, we introduced a constraint. We define a "forbidden zone" around the current pixel. Indeed, if the displacement is "too small" we consider it to be the same "object" and that it is not a copy.

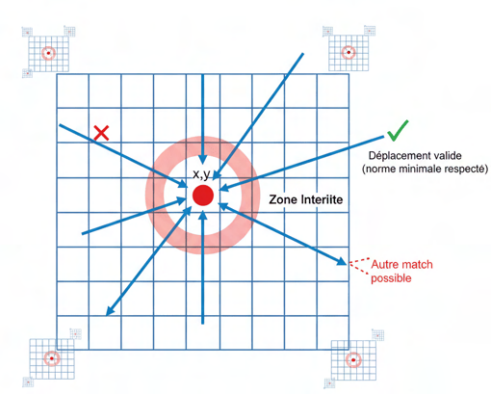


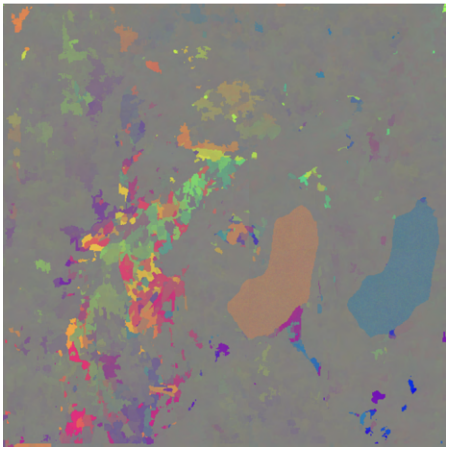
Figure 5: Illustration of the forbidden zone concept. The displacement vector must have a minimum norm.

This constraint is applied during initialization, propagation, and random search. If a candidate falls into this zone, it is rejected. This forces the algorithm to look for a match *elsewhere* in the image. If an area has been duplicated, the best "distant" candidate will be the duplicated area.

3.2 Displacement Field Visualization

To visualize the results, we cannot simply look at the reconstructed image (which would look like the original). We must visualize the offset vector field (dx, dy). We used a BGR visualization where the color indicates the direction of the displacement and the intensity its magnitude.

(a) Falsified original image



(b) Colored displacement field

Figure 6: Visualization of offsets. A uniform area of color indicates a coherent displacement (translation of an object).

We clearly notice that at the level of the copy-paste, the displacement zone is uniform and coherent. We also notice that on the contours of *lena* there are also displacements, which will have to be processed to be able to detect the copy-paste automatically. This is where we encountered our first interpretation difficulties. Uniform areas (sky, smooth walls) create noisy vector fields: a blue sky patch looks like any other blue sky patch. The displacement field is therefore "chaotic" in uniform areas, but very "smooth" (constant color) in copied textured areas.

4 Automatic Detection

Having a colored displacement map is useful for a human, but insufficient for automatic binary detection (copy-move / Non copy-move). This is the step that took us the most time. We first tried with classic thresholding, which does not give a good result, either too restrictive or not enough.

Then, using clustering algorithms like *k-means*, we tried to detect copy-paste areas in the displacement mapping. Again without much success.

We therefore did some research to try to find a new paper that would help us with automatic detection. We therefore relied on Ehret's article [2] to transform our noisy vector field into a clean detection mask.

The paper suggests several post-processing steps that we implemented in the second part.

4.1 Median Filtering

The raw vector field generated by *PatchMatch* is noisy, particularly in homogeneous areas where convergence is random, displacement vectors are isolated and incoherent. Before analysis, we apply a median filter on each component (dx and dy) of the field [2]. Unlike classic Gaussian smoothing, this non-linear filter allows eliminating impulse noise (isolated outliers) while preserving transition sharpness (sharp edges). This step is crucial to preserve the exact boundaries of the falsified object while cleaning the background.

4.2 Error Map

This is the core of the automatic detection. The idea is to measure the *local coherence* of the displacement. In an area copied by translation, all neighboring pixels must have approximately the same displacement vector. The local variance of the offsets must therefore be close to zero. Conversely, in a uniform natural area (sky), the vectors go in all directions (noise), the local variance is high.

We implemented a simplified version of the Error Filter[2], adapted to the translation case: This function generates an image where black pixels (low error) correspond to rigidly displaced areas (the copy-paste), and white pixels to incoherent areas.

- **Copied zone:** The local variance of vectors (dx, dy) is close to zero (all neighbors "point" in the same direction).
- **Homogeneous (natural) zone:** The local variance is very high (vectors are chaotic).

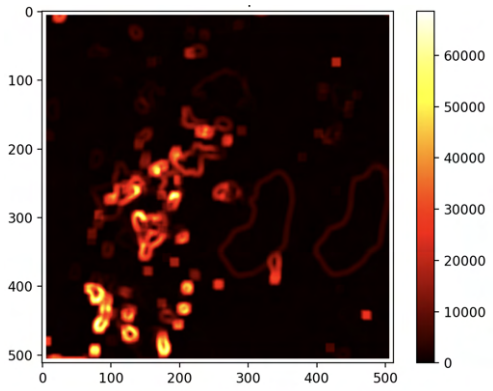


Figure 7: Error map (previous modified lena image)

Here we clearly notice that at the level of the copy-paste the variance is zero. Conversely, where the displacement mapping recognized something abnormal, while it was only contours, the variance is large on the error map.

4.3 Global Histogram Filtering and RMSE

To improve robustness against repetitive natural textures (like road tar or a brick wall), local coherence is not always enough. We introduced two new constraints in our pipeline to reduce false positives:

1. Global Frequency Filtering:

Instead of looking only at a pixel's neighbors, we analyze the entire image. In a natural texture, correspondences found by *PatchMatch* are scattered randomly. Conversely, a cloned copy implies that a large number of pixels (for example more than 1000 pixels) share exactly the same displacement vector (dx, dy). We therefore count the number of pixels that have the same displacement and remove those that do not have enough.

Figure 8: Visualization of the argument

2. Resemblance verification (RMSE):
A displacement vector can be geometrically coherent but visually wrong. We use the distance map provided by the algorithm to calculate the Root Mean Square Error (RMSE) of color between the source patch and the target patch (the pair of patches found by the algorithm). If this error exceeds a threshold (copy visually too different), the pixel is rejected.

4.4 Thresholding and Cleaning (Binary Mask)

Finally, to obtain the final mask, we combine these criteria via logical operations (intersection of masks):

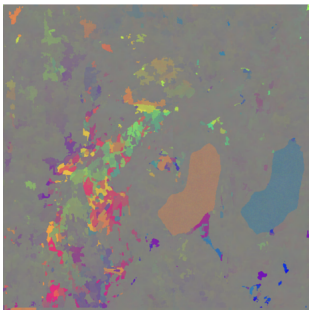
1. **Frequency filter:** We only keep pixels belonging to a massive displacement group (frequent vector).
2. **Error thresholding (Variance):** We only keep pixels having very strong local coherence ($err < \tau_{error}$).
3. **Visual quality (RMSE):** We eliminate false positives whose color resemblance is not sufficient.
4. **Min displacement threshold:** We eliminate pixels whose displacement is too weak (increasing the radius of the forbidden zone).
5. **Size filter:** We use `cv.connectedComponentsWithStats` to remove small pixel clusters (residual noise) and keep only significant shapes.
6. **Dilation:** We apply dilation to "densify" the detection, fill holes, and merge neighboring connected components.

Figure 9: Binary mask

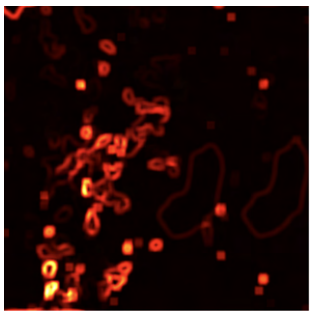
4.5 First Complete Result

In the figure below, which represents all steps of the copy-move detection, we directly obtain the binary mask, where white areas represent the copied area and the pasted area. Furthermore, the overlay allows identifying areas more simply. Here it is very obvious to see the falsification, we will see more concrete cases later.

(a) Falsified image



(b) Offset Mapping



(c) Error map

(d) Binary mask

(e) Overlay

Figure 10: Complete detection pipeline. (a) The input image containing a copy. (b) The dense vector field showing a coherent zone. (c) The statistical error map. (d) The overlay on the image directly.

5 Results and Discussion

We tested our algorithm on numerous images, with more or less obvious copy-moves, on blurred images, noisy images...

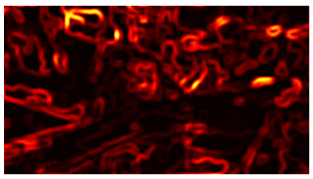
The results were very prolific for discussing the limits of this method.

5.1 Simple Translation

As seen on the falsified image of **Lena**, which is rather simple, images where an object has been duplicated by simple copy-paste (translation), the algorithm converges to a very satisfactory result.
In the figures below, we obtain results that are more than satisfactory, despite very homogeneous and smooth areas or high-frequency areas with many details.

(a) Falsified image

(b) Offset Mapping



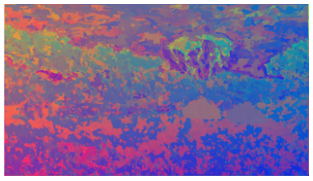
(c) Error map

(d) Binary mask

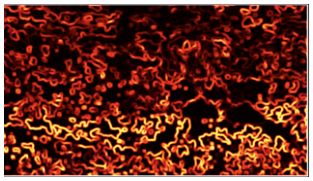
(e) Overlay

Figure 11: Trial on an image with a rather homogeneous road.

(a) Falsified image



(b) Offset Mapping



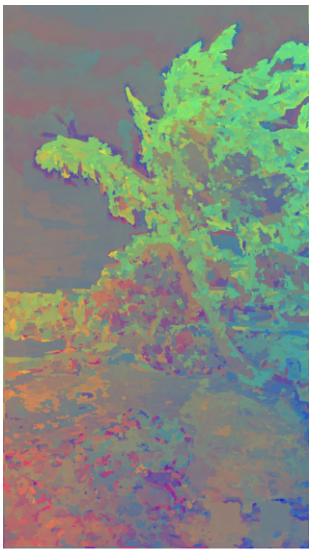
(c) Error map

(d) Binary mask

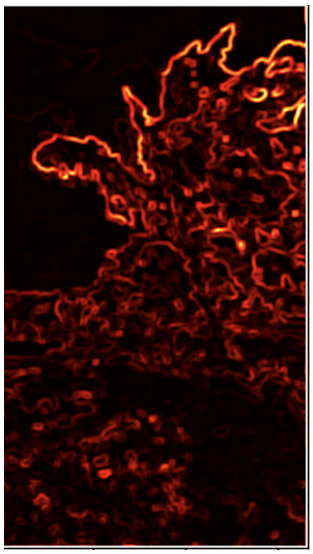
(e) Overlay

Figure 12: Numerous homogeneous zones (sky and tall grass).

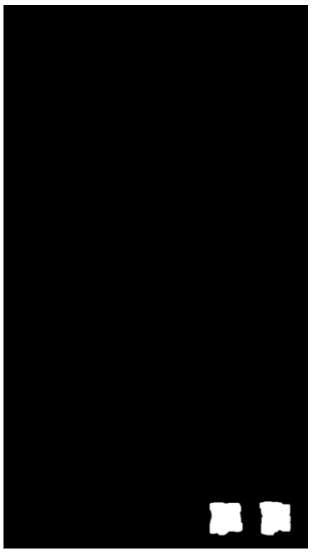
(a) Falsified image



(b) Offset Mapping



(c) Error map



(d) Binary mask

(e) Overlay

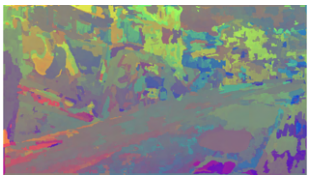
Figure 13: Photo with a lot of details and notably a very homogeneous sky.

5.2 Difficulties Encountered

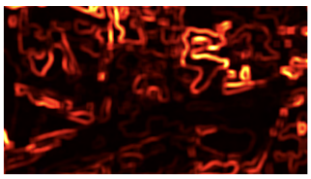
5.2.1 Uniform Areas and Highly Textured Areas

Previous trials gave a very suitable result, but this was not the case directly. Indeed, on these images, homogeneous areas, such as the sky, grass, or sea, are numerous, which can create false positives. A detection of a copy-paste that is not one, because a large area is similar. To overcome this problem, it was necessary to **increase** the radius of the **forbidden zone** "$R_{forbidden}$". The problem is the same for very high-frequency areas with many details, areas with repetitive textures (brick on a wall). With this repetitiveness, it creates a self-similarity of patches.

(a) Falsified image



(b) Offset Mapping



(c) Error map

(d) Binary mask

(e) Overlay

Figure 14: Limits encountered with a too low $R_{forbidden}$.

Uniform areas (road and wall) create many false positives, and the result is not satisfactory at all. This therefore allowed us to understand the impact and weight of certain coefficients.

However, when images are very simple, like Lena's, with a very visible copy-paste, having an $R_{forbidden}$ does not allow convergence towards a satisfactory result of the algorithm.

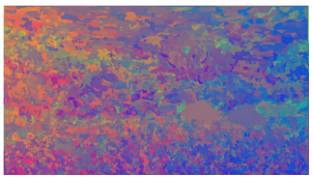
5.2.2 Parameterization

These numerous criteria created many parameters that need to be adapted according to the type of image and the type of copy-paste (small or large object for example). It is necessary to fully understand the utility of each of these parameters and the weight they have within the algorithm.

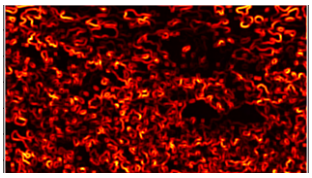
5.2.3 Blurred Images

Our method is very robust against blurred images, which is very important because it is a common defect in real photographs (motion blur, missed focus). This robustness is explained by the very nature of the algorithm: PatchMatch compares pixel neighborhoods (patches) and not isolated pixels. Blur acts as a low-pass filter that smooths details (high frequencies) but preserves the global structure and low frequencies of the image. Thus, the distance between a source patch and its blurred copy remains low enough for the correspondence to be detected by the algorithm.

(a) Falsified blurred image



(b) Offset Mapping



(c) Error map

(d) Binary mask

(e) Overlay

Figure 15: Significantly blurred images.

We notice that the base image is very blurred yet the result is perfect.

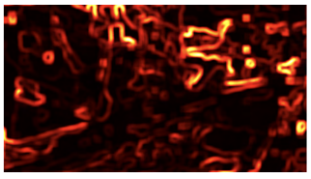
5.2.4 Noisy Images

Our method is rather robust to added noise, especially impulse noise which is very well handled by the **median filter**. Noise will have the effect of disturbing the distance D that we use to compare two patches. Noise will therefore pose the following problem: patch a will no longer look quite like patch b. For the different tests, we took ready-made programs to add noise to the falsified images. Then we run the algorithm on these images.

Gaussian Noise

(a) Falsified image

(b) Offset Mapping



(c) Error map

(d) Binary mask

(e) Overlay

Figure 16: Images noisy with Gaussian noise.

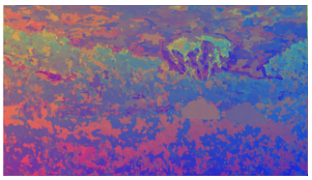
We notice that despite the added noise, our algorithm converges well towards a solution without "false positives".

Noise due to JPEG compression

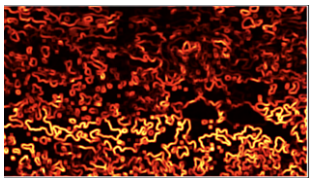
We notice that the algorithm is robust against noise due to JPEG compression, which was very important to us because there are many compressed images on

the internet.

(a) Falsified image



(b) Offset Mapping



(c) Error map

(d) Binary mask

(e) Overlay

Figure 17: Image noisy with compression noise.

5.3 Limits of this Method

5.3.1 Copy-Move in Uniform Areas

When the copy-paste is done in the smooth area, even by increasing the forbidden zone radius, this generates "false positives" because they possess strong natural self-similarity, even without falsification. The algorithm does not converge to an acceptable solution. It often finds nothing or an area that is not a copy.

(a) Original image

(b) Falsified image

Figure 18: Copy-paste in grass to hide a person.

5.3.2 Inpainting

For classic inpainting, copy-pasting known patches into the image, our algorithm will struggle to find the origins of the copied patches and where they are pasted. Indeed, it will take these copy-pastes as noise. We must try to make statistical filtering and the median less important to have a suitable result. The vector field will be much more chaotic, because the copied patches will not have the same origin.

To analyze how our algorithm reacts to inpainting, we asked the group composed of **Raphaël Legrand** & **Arthur Evain** for their algorithm to do this test. We thank them.

(a) Original Image (b) Inpainted Image (c) Detection Mask

Figure 19: Result of our algorithm on an inpainted image.

Inpainting, as on the road image previously, is used to hide the blue rectangle on the road. We notice firstly that details at the level of the gate and the sky are taken as good matches. This is rather logical, since we reduced statistical and median filtering, micro-textures as well as uniform areas are selected. We see however, that the algorithm still found patches on the road that were used.

5.3.3 Scaling and Rotation

Since the comparison is done patch by patch, pixel by pixel, it is rigid to scaling as well as rotations. The problem being in the norm we use, we will not compare the "same" pixels, because they will no longer have the same layout in the grid. The norm will be large and therefore even if visually the patches look alike, algorithmically they will not be associated. Invariant descriptors by translation and rotation are therefore needed [2], for the algorithm to be robust against these.

6 Conclusion

We finally made our own copy-move detection algorithm and obtained rather satisfactory results. Indeed, the results we obtain are in line with the theory: Inpainting, rotation, scaling. Furthermore, it is rather robust to the different problems that an image could encounter in real life (noise, blur, compressions). This project required a lot of time from us, especially to dive into a research paper for the first time, and we are very satisfied with this first experience.

As for the improvements we would have liked to implement: A GPU implementation which attracted us a lot initially, in order to parallelize the algorithm steps. However, the lack of time did not allow us to make this improvement, which would have been very instructive. Or robustness against rotation and scaling, notably using Zernike descriptors [2]. These descriptors allow representing the content of a patch independently of its orientation.

Finally, this project allowed us to learn more about image processing, particularly the *Patchmatch* algorithm which is widely used in this field. We thank Yann Gousseau for his explanations on this subject and for answering the many questions we asked him.

References

[1] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. *PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing.* ACM Transactions on Graphics (TOG), 28(3), 2009.

[2] T. Ehret. *Automatic Detection of Internal Copy-Move Forgeries in Images.* Image Processing On Line, 9:47–77, 2018.