Image-to-Image Steganography for Watermark Creation Report

COM31006: Computer Vision Coursework

Nirmal Eswar Vee
University of Sheffield

May 27, 2025

1 Functions

This section will explain the logic behind the functionality of the source code as well as the rationale behind specific design choices. The main functions in this report include:

- Detect Keypoints
- Embed Watermark
- Recover Watermark
- Detect Tampering

1.1 Detect Keypoints

Keypoint detection can be split into several different steps to achieve high saliency and spatial coverage:

1. Instantiate the SIFT detector using the `cv2.SIFT` library.
2. Detecting all keypoint objects using `SIFT.detectAndCompute`, which returns a list of keypoint objects as well as an array of the descriptors.
3. Extract the response scores (measures how salient the keypoint is) and coordinates from each keypoint and put them into two separate NumPy arrays
4. Sort the keypoints by how salient they are by using `np.argsort` on the response score array.
5. For each coordinate in the coordinates array, compute its grid cell and store them in an array of cells. This is done to avoid all the features with high response scores clustering into one cluster.
6. From each grid cell, select the top keypoints (using the order in the sorted response score array).
7. If fewer than `max_kps` have been selected, "fill" the remaining slots by taking the next strongest keypoints from the globally sorted list (skipping any already chosen).
8. Add these to the selected keypoints list and return the final list of selected keypoint objects in the order you've selected.

These selected keypoints are then returned, signifying the most salient and spatially well-distributed feature locations in the image. I chose to store the responses and coordinates in the NumPy arrays as opposed to lists to exploit how NumPy arrays can be vectorised, allowing for quicker processing times compared to using lists for everything.

1.2 Embed Watermark

In order to embed the watermark, I followed these steps:

1. Copy the input greyscale image into a NumPy array
2. Turn the watermark into a binary array of shape $(H_{\text{wm}}, W_{\text{wm}})$.
3. For each keypoint:
   a) Compute watermark pixel indices (u, v) via

$$u = i \bmod W_{\text{wm}}, \quad v = \lfloor i / W_{\text{wm}} \rfloor$$

b) Loop over the first H_{wm} * W_{wm} bits:

i. Extract the bit $b = \texttt{watermark}[v, u]$.

ii. Replace the pixel at (y, x) so that it becomes even if $b = 0$ or odd if $b = 1$.

4. Return this modified image array.

By following these steps, the function is able to output the image that is passed into it as a modified version with a watermark.

1.3 Recover Watermark

A function to recover the watermark was also implemented in the following way:

1. Redetect the same keypoint in the (possibly watermarked or tampered) input greyscale image using identical SIFT + grid logic that was used to embed the watermark.
2. Unpack the original dimensions (height and width) of the watermark because it's been passed in as a parameter.
3. Then each keypoint's index up until H_{wm} * W_{wm} is mapped to a watermark pixel coordinate.
4. Read the least significant bit (LSB) of the image pixel and store it in an array of recovered bits.
5. Return the watermark array which contains the recovered watermark bits.

This function allows the user to get a matrix of the watermark bits exactly as they were embedded by effectively reversing the process described in subsection 1.2.

1.4 Detect Tampering

The detect tampering function detects if the watermark data at the currently detected keypoint locations differs from the original watermark:

1. First, the bits and keypoints of the watermark are recovered by calling recover_watermark.
2. Each recovered bit of the watermark is compared to the corresponding original watermark bit.
3. Wherever they don't match are the tampered points, and they are stored in a tampered points array.
4. The tampered points array and the recovered watermark array are then returned.

By comparing each recovered bit against the original watermark bits, the function shows regions in the image where the embedded watermark data has been modified.

2 UI

The core functionality outlined in the previous section is wrapped in a UI class called WatermarkApp. The WatermarkApp class that I created contains three different functions (embed, authenticate and detect), which each correspond to their own UI button and call the appropriate functions that were previously implemented. It also contains a parameter frame where the user can change the parameters being used. The complete UI frame is shown in the image below.

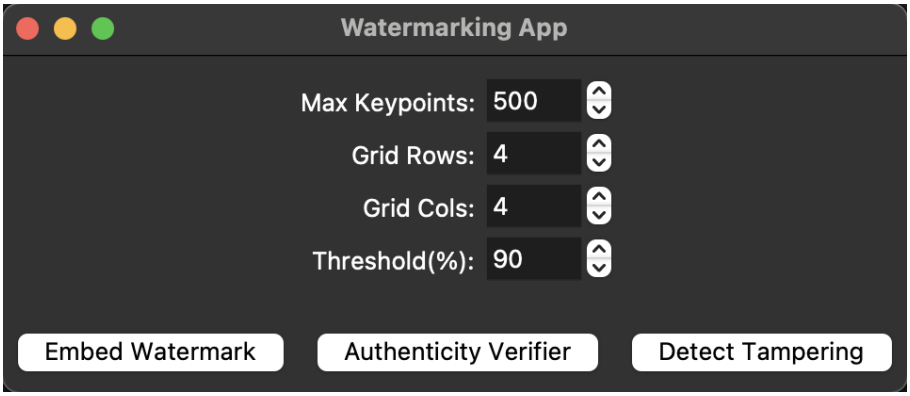

Figure 1: UI frame

Below I have included a class diagram for the WatermarkApp class I created in Graphviz to help aid understanding.

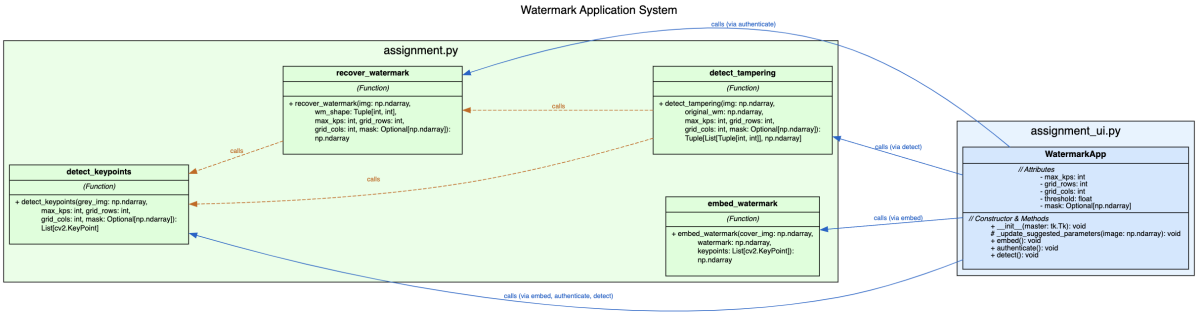

Figure 2: Class diagram made in GraphViz

In the following subsections I detail each UI button and what functionality they call upon.

3 The buttons

All the buttons follow a similar workflow of:

1. Opening a file picker to prompt the user to open the specified images.

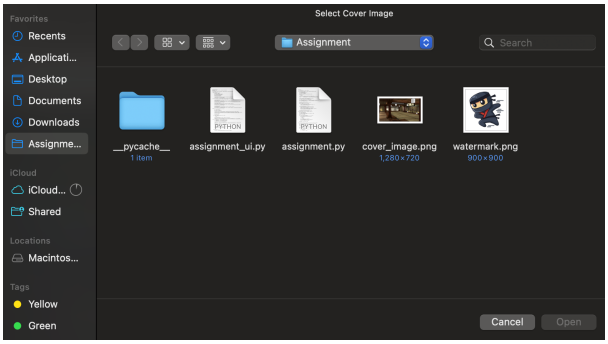

Figure 3: Select cover image

2. Both files are then read in greyscale, where the parameters are updated to reflect the dimensions of the image supplied by the user (see section 4).

3. Adaptive thresholding (`cv2.adaptiveThreshold`) is used to convert the watermark into a binary array depending on if there is an alpha channel or not (see section 5).

4. The function detect_keypoints is called to detect all the keypoints in the image, and the watermark is dynamically resized (see section 5).

5. This step is now different for each button.

3.1 Embed button

Embed button embeds the watermark onto an image. Steps 1-4 are followed. Step 5 is the LSB-based embedding, which is applied to each keypoint by calling embed_watermark, producing the watermark-embedded image. The user is then prompted to save the watermarked image under a name.

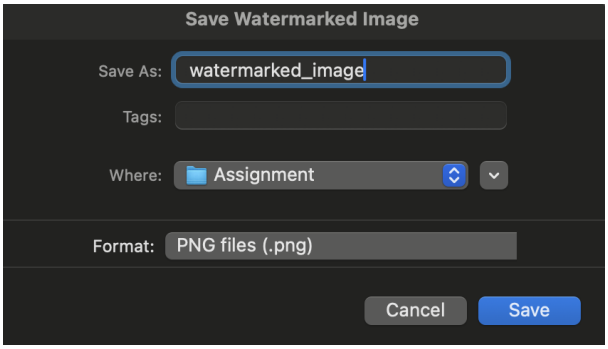

Figure 4: Saving the image

The watermarked image is then saved under this name in the appropriate directory.

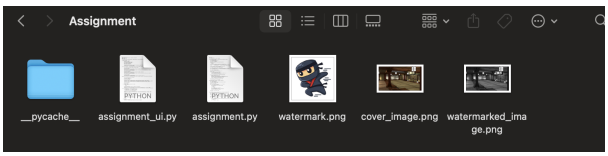

Figure 5: Watermarked image saved in appropriate directory

3.2 Authenticity Verifier button

The authenticity verifier button checks if the selected watermarked image has the original watermark intact. Steps 1-4 are followed, with step 1 asking for the watermarked image and original watermark. Step 5 is recover_watermark being called to re-detect the same SIFT keypoints and read the LSB at each keypoint location by passing in the updated grid parameters. The binary array of the original watermark and the recovered bits at the keypoint locations are then compared against a threshold value to see if the original watermark is intact or not.

3.3 Detect button

The detect button checks if the watermarked image has been altered or tampered with, and if it has, it returns an image with the tampered keypoint locations highlighted. Steps 1-4 are followed, step 1 asking for the image to test and the original watermark. Step 5 is detect_tampering being called to find the tampered locations and recovered bits in the test image. If the percentage of matching bits falls below the threshold, an image where red circles are drawn at the tampered locations is returned.

4 Parameters

The user can adjust the parameters in the parameter frame of the UI box.

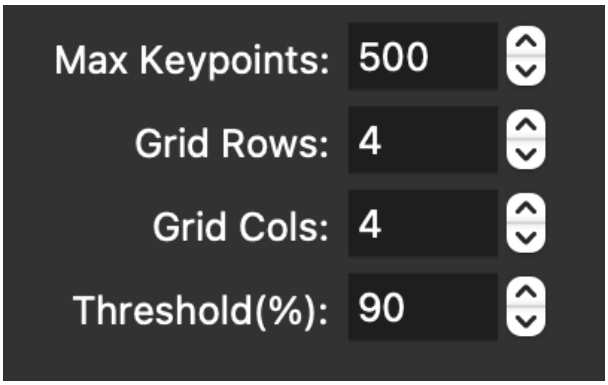

Figure 6: Parameter Frame

The role of each parameter and the effect that comes from changing them are as follows:

- **Max Keypoints:** Sets the total number of SIFT keypoints used to embed watermark bits.
  - Increasing the max keypoints results in higher resolution watermarks but more total pixel changes.
  - Decreasing the max keypoints results in smaller watermarks and potentially poorly distributed bits if the value is too low.
- **Grid Rows and Columns:** Partitions the image into a grid of buckets for keypoints to live in.
  - The number of rows and columns doesn't need to be equal, but it helps if they are roughly the same so the grid is roughly a square that ensures each bucket covers a similar area.
  - The effect of increasing either dimension is a more uniform distribution but fewer bits per cell.
  - The effect of decreasing either dimension is more bits in each cell, so if a patch survives, more bits can be recovered, but there is also the risk of losing a lot more bits if the patch doesn't survive.
- **Threshold(%):** Defines the minimum percentage of matching bits between the recovered watermark and the original (resized) watermark that is required for the watermark to be considered "intact".
  - Increasing the threshold makes the verification process stricter.
  - Decreasing the threshold makes the verification process more lenient.

4.0.1 Automatic Parameter Suggestion

The app automatically suggests default values for the Max Keypoints, Grid Rows, and Grid Columns parameters. These initial values are updated whenever the first image is uploaded by the user when pressing any of the function buttons. These suggestions are computed from the loaded image's properties:

- The Max Keypoints value is calculated based on the image's total pixel area, aiming for roughly one per thousand pixels whilst also sticking to the predefined minimum and maximum limits set in the UI.
- The grid rows and grid columns are determined by the image's height and width, around 200 pixels per side, again respecting the UI limits.

These dynamically calculated values automatically show up in their corresponding fields in the Parameter Frame (Figure 6). These are just a suggestion, as users can still adjust these parameters to their liking.

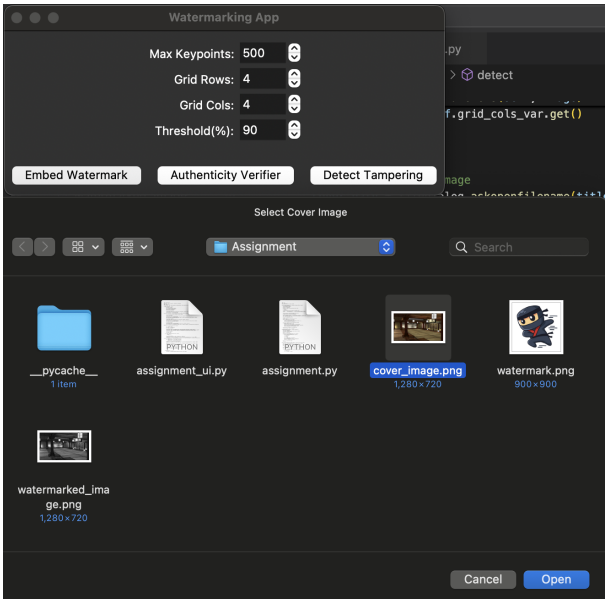

Figure 7: The parameters before choosing the cover image

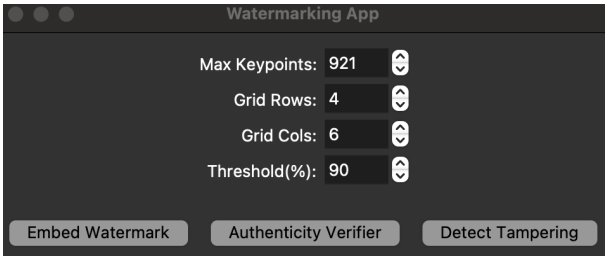

Figure 8: The parameters automatically adjusted after choosing the cover image

5 Innovation

I included innovative features to help with my watermark embedding, authentication and tampering detection. They are as follows:

- **Grid-Based Keypoint Selection for Enhanced Spatial Distribution:** The image is divided into a user-defined grid using the grid rows and columns parameters, and strong keypoints are chosen from each cell. This spreads the watermark across the whole image, making it better equipped to deal with local edits like cropping.
- **Alpha Channel Processing for Watermark Definition:** The alpha channel is extracted, and adaptive thresholding is applied to create a binary mask. With this feature

any image with transparency could be used as a watermark. This gives the app more robustness.

- **Dynamic Watermark Resizing:** After detecting how many SIFT keypoints are available, the watermark is automatically scaled down, using nearest neighbour to exactly fill the slots for the number of keypoints detected. This means the watermark never overflows or loses bits during embedding or comparison.

With these more innovative features that deviate from the standard approach outlined in the brief, the app becomes more robust.

6 Results and Evaluation

This section discusses the results and how well the app performed. The images that the tests are performed on are these:

Figure 9: The original watermark

Figure 10: The cover image to have the watermark embedded on

6.1 Embedding the watermark test

Figure 11: The watermark embedded image

There is no visual distortion, and the watermark is hidden, so this button passed the functionality test.

6.2 Authentication verification test

When passing in the watermarked image (figure 11)

Figure 12: Authentication passed

When passing in an image that doesn't contain the original watermark:

Figure 13: Image without the embedded watermark

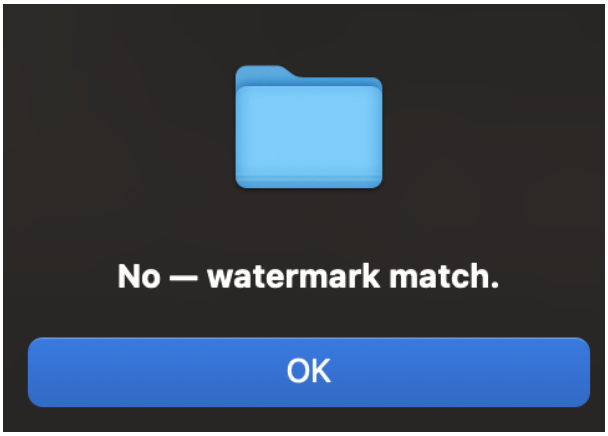

Figure 14: No watermark match

Authenticator correctly identifies when the watermarked image is passed in or not so it passed the functionality test.

6.3 Detection tampering test

When passing in the watermark embedded image (figure 11)

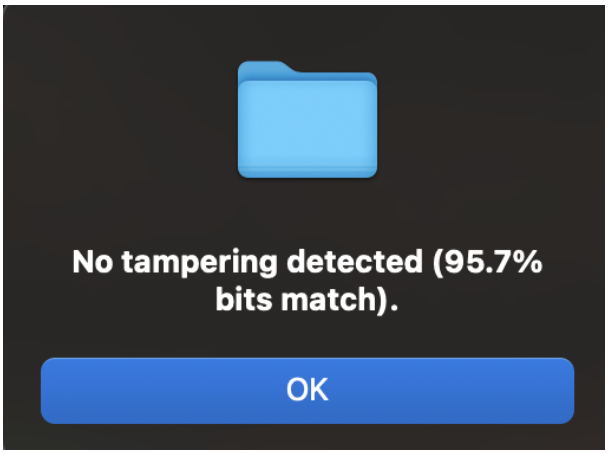

Figure 15: No tampering detected

No tampering is detected. When passing in a tampered image (the watermarked image rotated):

Figure 16: Rotated watermarked image

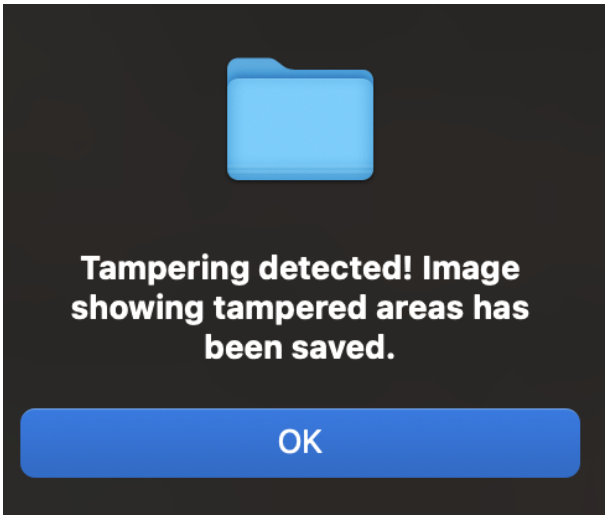

Figure 17: Tampering detected

Tampering is detected, and the tampered points are highlighted and saved as an image:

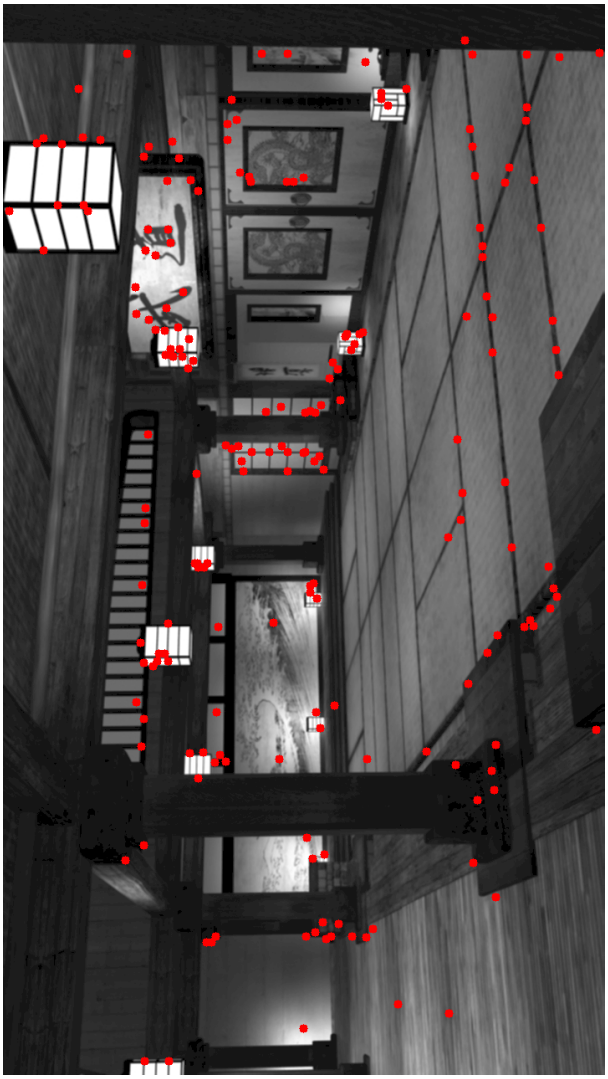

Figure 18: Tampered points highlighted

The detect_tampering button also passes the functionality test, as it can detect when the image hasn't been tampered and when it has, it returns an image showing the tampered locations.

7 Q&A, Limitations and Future Work

This section address key questions, limitations and future work.

7.1 Q&A

- Overlapping keypoints: Each bit is written to the LSB of a single pixel at the integer-rounded keypoint coordinate. If two keypoints ever coincide (rare), the later write simply overwrites the earlier one. Our grid-based selection spreads keypoints evenly and thus minimises such collisions.
- Deviations from the spec are outlined and my most innovative features are detailed in my innovations section 5
- Tampering detection effectiveness: Our LSB-at-SIFT approach reliably flags gross modifications (e.g. rotation, cropping), but is sensitive to JPEG compression, noise, and large

geometric changes that shift keypoints. Future work could add ECC, domain-transform embedding (DCT/DWT), watermark encryption, or spatially varying codes.

7.2 Limitations and Future Work

This implementation is functional but still has limitations, such as how LSB modifications are fragile and sensitive to noise. Secondly, the security is basic, as the watermark is not encrypted. The system also relies on redetecting keypoints using SIFT, but changing the image can prevent SIFT from finding these keypoints. For future work, I would explore embedding in transform domains instead of LSB and how to encrypt the watermarks. Also, as suggested in the brief, it would be worth exploring methods that vary watermark information spatially.

8 Link to Demonstration Video

https://youtu.be/gGkwXMf4hUE