PatchMatch

Eswar Kumar Anantha
2023102032
IIIT Hyderabad
anantha.kumar@students.iiit.ac.in

Duggi Yaswanth
2022102051
IIIT Hyderabad
Duggi.Yashwanth@students.iiit.ac.in

Introduction

Modern image editing tasks such as retargeting, texture synthesis, and object removal often rely on patch-based methods, where images are decomposed into small regions that can be rearranged or synthesised. However, traditional patch search is computationally expensive due to the need to compare a large number of patches across the image.

The PatchMatch algorithm addresses this challenge by efficiently estimating a Nearest Neighbour Field (NNF) through random search and propagation. This makes patch-based manipulation significantly faster and has made PatchMatch a widely used technique in image completion and visual reconstruction.

In this project, we reimplement the NNF algorithm entirely in Python, taking guidance from the original MATLAB and C++ reference implementations. We also develop a simplified image in-painting framework that combines an Expectation–Maximisation (EM) step with a Multi-Scale Coarse-to-Fine refinement strategy, inspired by an existing C++ implementation. Additionally, we provide a simple Python GUI that allows users to load images, draw mask regions, and export them for use in the in-painting pipeline.

PatchMatch

PatchMatch is an efficient algorithm designed to accelerate the search for similar image patches, achieving speedups of 20–100× compared to traditional exhaustive matching methods while requiring significantly less memory. The key insight behind PatchMatch is that neighbouring image patches often share strong statistical coherence, allowing the algorithm to rapidly propagate good matches across the image through a combination of random search and local optimisation. As a result, PatchMatch can compute dense nearest-neighbour fields—typically one patch match per pixel—with high accuracy and efficiency.

This capability makes PatchMatch particularly suitable for tasks such as image in-painting, retargeting, and texture reshuffling, all of which are explored in this project. The algorithm's effectiveness has led to widespread adoption, including integration into Adobe's professional image and video editing tools. Notably, the Content-Aware Fill feature in Photoshop and After Effects relies on PatchMatch to perform high-quality object removal and image completion by synthesising plausible content from surrounding textures.

Overall, PatchMatch has become a foundational component in modern computational photography due to its combination of speed, simplicity, and versatility.

Methodology

Image Details

Natural images exhibit two important statistical properties that enable efficient patch-based search: *coherence* and a *peaked distribution*. Coherence reflects the observation that neighbouring patches tend to share similar nearest neighbours; therefore, a good match found for one patch can often be propagated to adjacent patches. The peaked distribution property indicates that the best matches for a patch are frequently located close to its original spatial position within the source or target image.

By exploiting these characteristics, algorithms such as PatchMatch can focus their search on local neighbourhoods while propagating high-quality matches across nearby pixels. This significantly reduces computational cost and improves the speed and accuracy of nearest-neighbour field estimation. The following images illustrate these properties in practice.

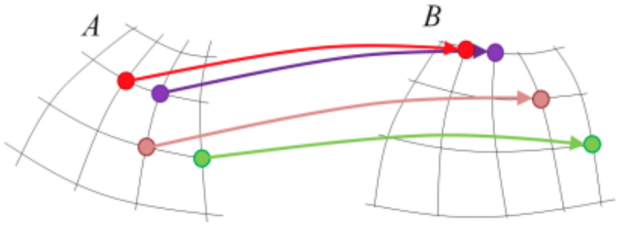

Fig. 1. High level motivation behind the algorithm. We match descriptors computed on two manifolds, visualised as colour circles. Each descriptor independently finds its most similar match in the other manifold. Coherence here could be indicated by the red descriptor being similar to the purple descriptor, and that their nearest neighbours are spatially close to each other

Nearest Neighbour Field (NNF) Algorithm

At the core of the system is an efficient method for computing patch correspondences between two images. The

objective is to determine, for each patch in one image, the most similar patch in another image. To achieve this, we define a Nearest-Neighbour Field (NNF), which is a function $f : A \to \mathbb{R}^2$ that maps every patch in image A to a corresponding patch location in image B.

Formally, for a patch centred at coordinate a in image A, the NNF identifies the nearest-matching patch in image B at coordinate b. Thus, the mapping is expressed as:

$$f(a) = b.$$

This function effectively encodes, for every pixel-level patch in the source image, the position of its best-matching counterpart in the target image.

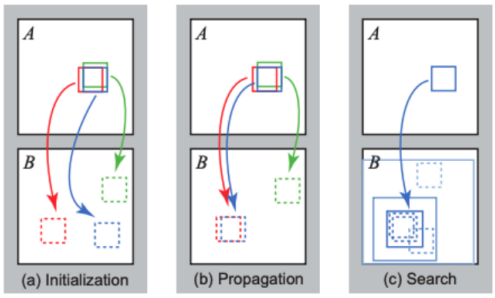

Fig. 2. Phases of the randomised nearest neighbour algorithm: (a) patches initially have random assignments; (b) the blue patch checks above/green and left/red neighbours to see if they will improve the blue mapping, propagating good matches; (c) the patch searches randomly for improvements in concentric neighbourhoods.

Initialisation

The initialisation phase populates the nearest-neighbour field with an initial set of patch correspondences. This can be performed in two primary ways:

Random Initialisation: Each patch in image A is assigned a random coordinate within the valid range of image B. These coordinates are sampled uniformly over the entire image, ensuring broad coverage of the search space and enabling the algorithm to discover diverse candidate matches.

Prior-Based Initialisation: In certain applications, prior information can be leveraged to produce a more informed initial NNF. For example, in our in-painting pipeline, the NNF at a finer resolution is initialised using the upsampled NNF from a coarser scale. This multi-scale strategy accelerates convergence and improves stability during refinement.

Once the initial NNF is constructed, the algorithm iteratively improves the offsets through two key steps: *Propagation* and *Random Search*. These operations are interleaved at the patch level, allowing the system to spread high-quality offsets from neighbouring patches while simultaneously exploring nearby candidate matches for further refinement.

Propagation

Propagation is a key refinement step in the NNF algorithm, where good offset estimates are transferred to neighbouring patches. For a patch located at coordinates (a_x, a_y), the algorithm assumes that its nearest neighbour is likely to be similar to the nearest neighbours of adjacent patches at $(a_x - 1, a_y)$ and $(a_x, a_y - 1)$. This assumption is motivated by the local coherence of natural images, in which neighbouring patches often exhibit similar structures.

To refine the offset $f(a_x, a_y)$, the algorithm compares its current match with the offsets propagated from its neighbouring patches. Formally, this can be expressed as:

$$f(x, y) = \arg\min \{D(f(x, y)),\ D(f(x-1, y)),\ D(f(x, y-1))\},$$

where $D(f(x, y))$ denotes the patch distance (error) between the patch centred at (x, y) in image A and the patch located at $f(x, y)$ in image B. This distance is computed as the squared RGB difference over all pixels in the patch.

Through this process, regions that already contain high-quality matches naturally propagate those matches to adjacent patches, improving the accuracy of the NNF across continuous image regions.

To avoid directional bias, the algorithm alternates its scan order. During odd iterations, propagation proceeds left-to-right and top-to-bottom as described. During even iterations, a reverse scan is performed, allowing propagation from the offsets at $f(a_x + 1, a_y)$ and $f(a_x, a_y + 1)$, thereby spreading good matches upward and leftward. This bidirectional propagation ensures more uniform refinement across the image.

Random Search

Following propagation, the algorithm performs a random search to further refine the offset for each patch. Starting from the current offset estimate, the algorithm explores nearby candidate offsets by introducing random perturbations. These candidate offsets are generated according to:

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

where v_0 is the current offset, w is the maximum search radius, α is a decay factor (typically set to $\frac{1}{2}$) that progressively reduces the search radius, and R_i is a random vector whose components are uniformly sampled from the interval $[-1, 1]$. This ensures that the search explores both horizontal and vertical directions around the current estimate.

During this process, the algorithm evaluates each candidate offset by computing its patch distance. If a candidate produces a lower matching error than the current offset, it is accepted as the new best offset.

The random search proceeds in an exponentially shrinking pattern: the search radius $w\alpha^i$ decreases at each step until it falls below one pixel. At this point, the neighbourhood around the patch has been sufficiently explored, and the search terminates. This multi-scale random exploration allows the algorithm to efficiently escape local minima while still converging to accurate nearest-neighbour matches.

Other Implementation Details

1) Evaluation Criteria: Each candidate offset generated during propagation or random search is evaluated using a patch distance metric based on the ℓ_2-norm of the RGB differences between the patches. If a candidate produces a smaller distance than the current match, the NNF is updated accordingly. When a patch contains masked regions, the distance is penalised by adding a large constant to discourage selecting matches that overlap with missing or invalid areas.

2) Halting Criteria: The refinement process is repeated for a fixed number of iterations, typically between four and five, until the NNF stabilises. Convergence is reached when additional iterations no longer produce meaningful improvements in nearest-neighbour assignments.

3) Patch Width: A patch width of 7 pixels is used in our implementation. This size offers a practical balance between capturing relevant texture information and maintaining computational efficiency. Using smaller patches may fail to represent broader structures, while larger patches increase computational cost and risk introducing excessive smoothing.

In-painting Application

We implement image in-painting using the Nearest Neighbour Field produced by the PatchMatch algorithm. The method operates within a multi-resolution coarse-to-fine framework and incorporates an Expectation–Maximisation (EM) procedure to iteratively update and refine the missing regions.

Initialisation

The in-painting pipeline begins with the initialisation of the key inputs: the colour image (represented as a 3D array), the binary mask identifying known and missing pixels, and several hyperparameters such as the patch size (`patch_w`) and the maximum number of PatchMatch iterations (`max_pm_iters`). The mask restricts the reconstruction process to the designated missing regions, ensuring that only those areas are synthesised.

A crucial component of the initialisation step is the construction of an image pyramid. This pyramid contains progressively downsampled versions of both the image and the mask and enables a coarse-to-fine optimisation strategy. At lower resolutions, the algorithm captures global structural information, providing stable initial estimates for large missing areas. As the resolution increases, the algorithm progressively refines texture details, leading to a more coherent and visually consistent reconstruction in the final output.

Pyramid Construction

The image and mask are progressively downsampled by a fixed factor (e.g., 2) until the smallest resolution becomes comparable to the patch size. During downsampling, the `downsample_img_and_mask` method applies Gaussian smoothing using a custom kernel to reduce noise and ensure smooth transitions across pyramid levels. The mask is downsampled by averaging weighted contributions from neighbouring pixels, allowing missing regions to be proportionally preserved at each scale. This multi-resolution pyramid forms the foundation of the coarse-to-fine in-painting strategy, beginning from the coarsest level.

EM Optimisation

The Expectation–Maximisation (EM) procedure alternates between estimating pixel values (E-step) and updating them (M-step) within masked regions.

E-Step: For each missing pixel, weighted contributions from overlapping patches are aggregated to form a set of "votes." For a patch centred at coordinates (a_x, a_y) in the source image and (b_x, b_y) in the target image, the weight w representing the similarity between the two patches is computed as:

$$w = 1 - \frac{d(P_A, P_B)}{\text{MAX\_PATCH\_DIFF}},$$

where $d(P_A, P_B)$ denotes the patch distance. These weights are normalised and used to accumulate pixel estimates based on the similarity of neighbouring patches.

M-Step: Pixel values in the masked region are updated by averaging the weighted votes obtained in the E-step. This iterative refinement gradually improves the reconstructed region, ensuring that the in-painted pixels are consistent with the surrounding context.

Multi-Scale Coarse-to-Fine Refinement

The refinement process begins at the coarsest pyramid level, where global structures and large-scale patterns are addressed. Missing regions are first estimated using PatchMatch and improved through several EM iterations. Once convergence is achieved at this scale, the reconstructed result is upsampled to the next level using bilinear interpolation, maintaining structural continuity.

The NNF computed at the coarser level is also upsampled and used to initialise PatchMatch at the finer level. This significantly reduces computational overhead and preserves coherence across resolutions. The process is repeated for each pyramid level, progressively reconstructing finer textures and details until the original image resolution is reached.

Output

At the highest resolution, the in-painted image contains reconstructed regions that are visually consistent with the surrounding content. The combination of PatchMatch for texture synthesis and EM-based refinement ensures that the final output exhibits both global structural consistency and high-quality local texture reconstruction. Mathematically, the multi-scale optimisation converges toward a solution that minimises patch distance across pyramid levels, producing a natural and coherent in-painting result.

Image In-painting

The mask required for in-painting can be generated using the mask creation tool provided in `get_mask.py`. Once the mask is obtained, the in-painting algorithm begins by constructing a multi-resolution pyramid of both the image and the mask through successive downscaling.

At each pyramid level, the Expectation–Maximisation (EM) routine is executed multiple times to iteratively reconstruct missing regions and refine the target image. After convergence at a given resolution, the reconstructed result is upsampled and used as the initialisation for the next finer level. This coarse-to-fine refinement continues until the highest resolution is reached, producing the final in-painted image that is consistent in both structure and texture. Results must store

Fig. 4. Visualization of the multi-scale in-painting pipeline on the hand tattoo image

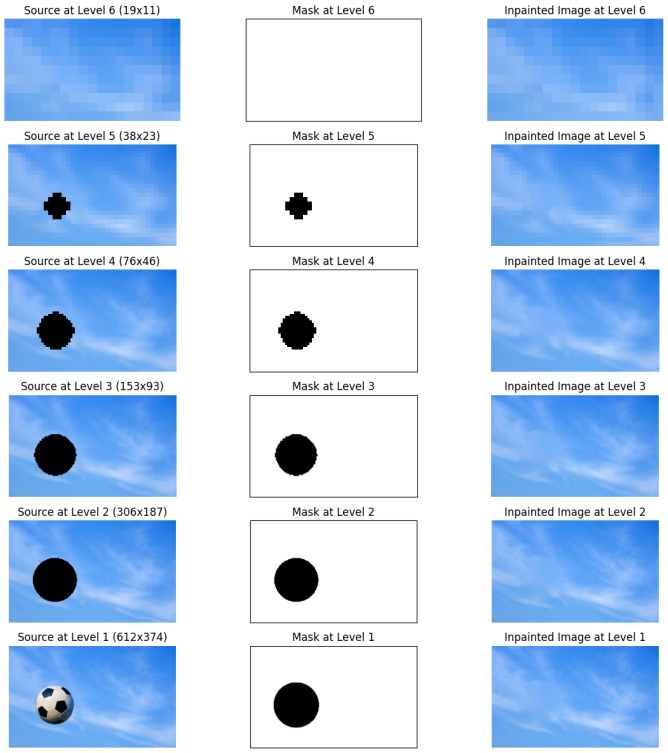

Fig. 3. Visualization of the multi-scale in-painting pipeline on the Football image

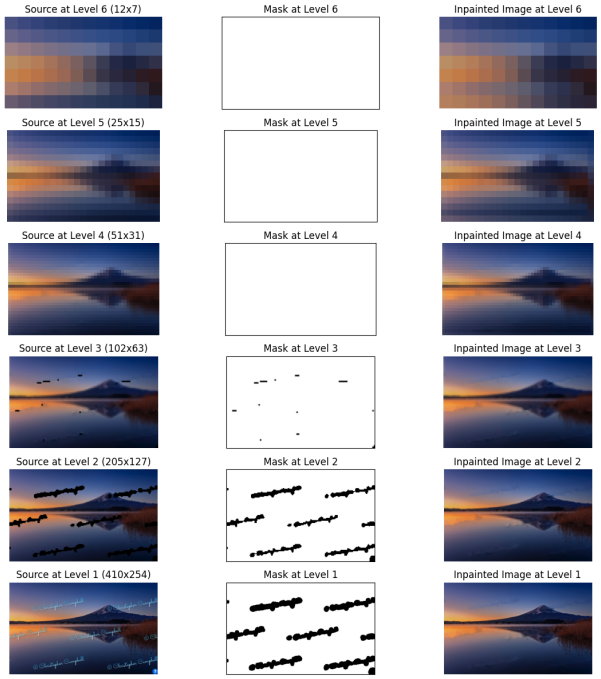

Fig. 5. Visualization of the multi-scale in-painting pipeline on the Corupted sign image

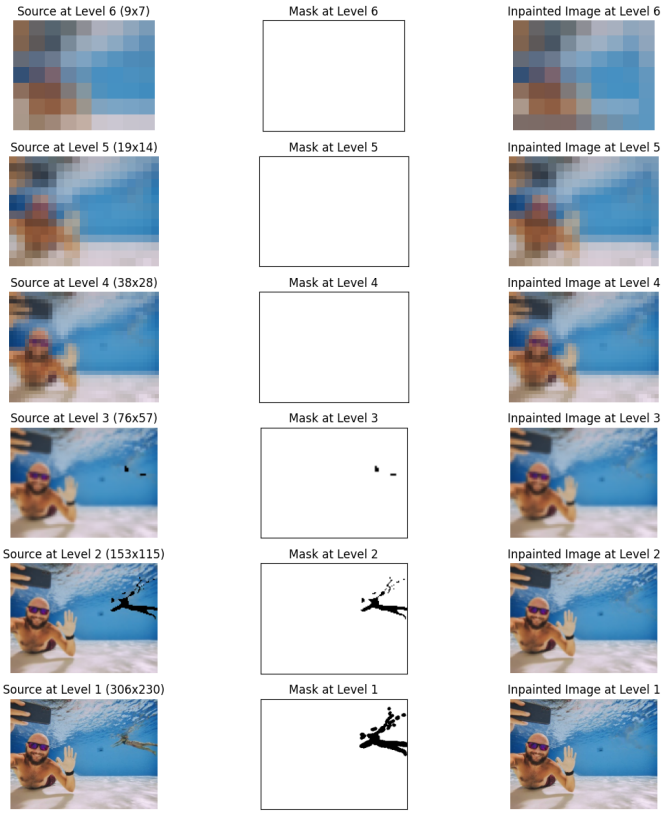

Fig. 6. Visualization of the multi-scale in-painting pipeline on the man under water image

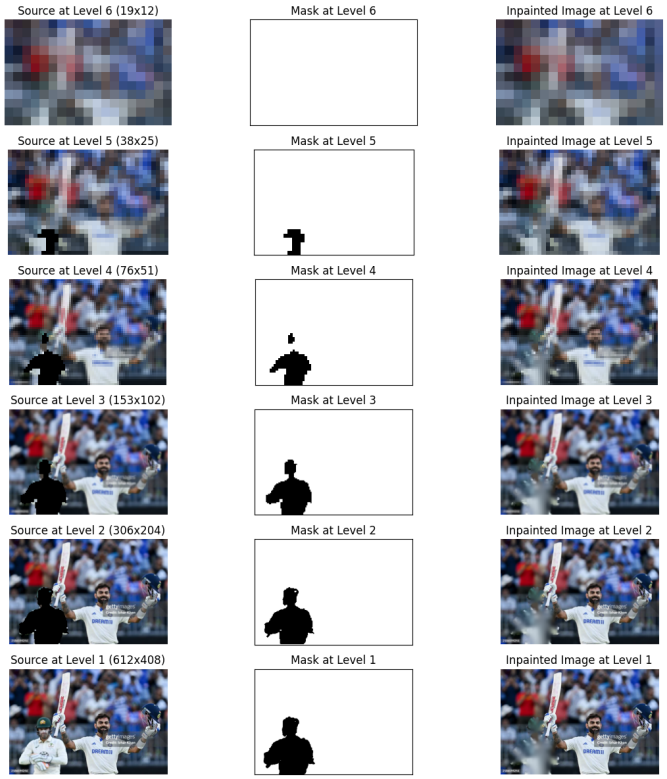

Fig. 8. Visualization of the multi-scale in-painting pipeline on virat image

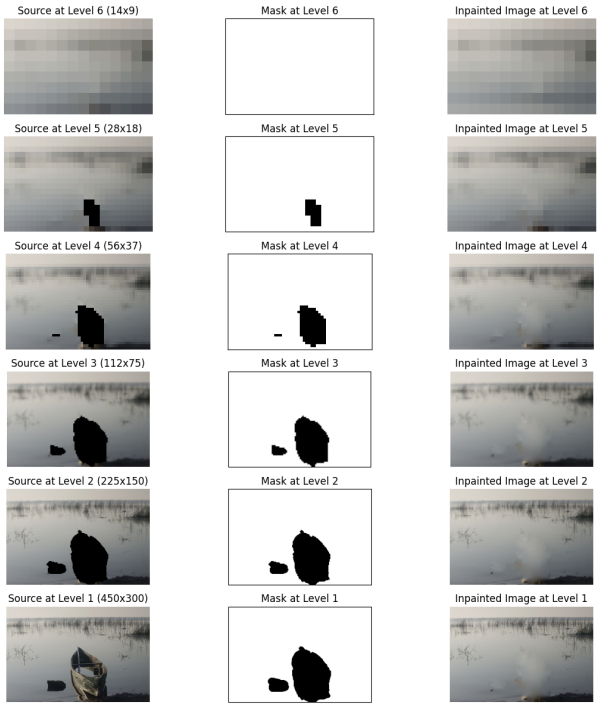

Fig. 7. Visualization of the multi-scale in-painting pipeline on the boat in lake image

Fig. 9. Visualization of the multi-scale in-painting pipeline on man standing in wall image

Ablation Study

To analyse the contribution of each component of the PatchMatch algorithm, we conduct an ablation study evaluating the effectiveness of the propagation and random search steps in Nearest Neighbour Field (NNF) computation for the task of image in-painting. We consider the following three configurations:

1) **Propagation + Random Search:** The full PatchMatch algorithm, serving as the baseline.
2) **Random Search Only:** Propagation is disabled, allowing us to assess the impact of pure stochastic exploration.
3) **Propagation Only:** Random search is removed, isolating the effect of local coherence-based refinement.

These configurations are evaluated on a single test image to observe how each component influences reconstruction quality, convergence behaviour, and the ability to recover fine textures and global structure.

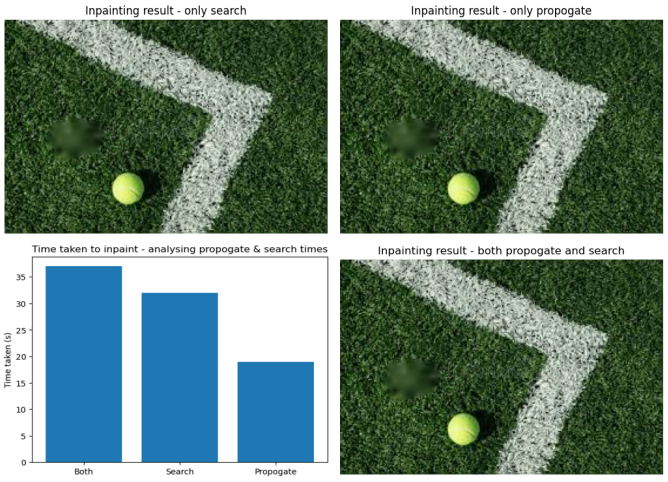

Fig. 10. sample Image 1 demonstartes 3 configurations

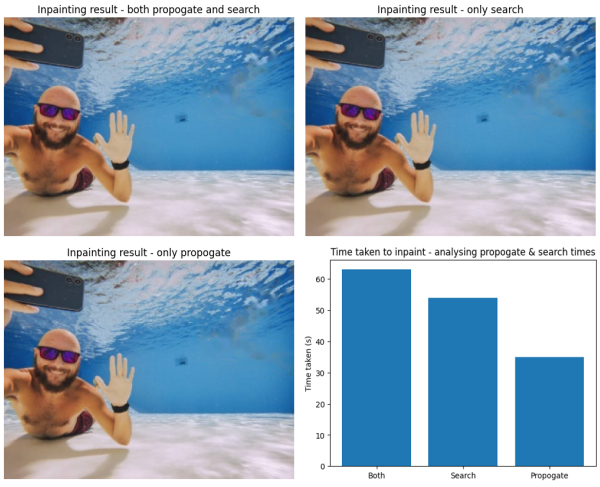

Fig. 11. sample Image 2 demonstartes 3 configurations

Image Reshuffling

Image reshuffling enables the user to reposition or rearrange regions within an image while automatically synthesising the remaining content so that the final output remains visually coherent. The goal is to preserve the overall appearance of the original image while accommodating the newly moved regions.

To perform image reshuffling using our in-painting framework, the desired region is first copied from its original location and pasted at a new position within the same image, producing an edited intermediate image. A corresponding mask is then constructed to remove the original object and to smooth the transition between the pasted region and the surrounding background. This mask is applied to the edited image during in-painting, allowing the algorithm to synthesise new content that fills the gaps and blends the boundaries seamlessly. The final result is a reshuffled image that maintains structural and texture consistency across the modified regions.

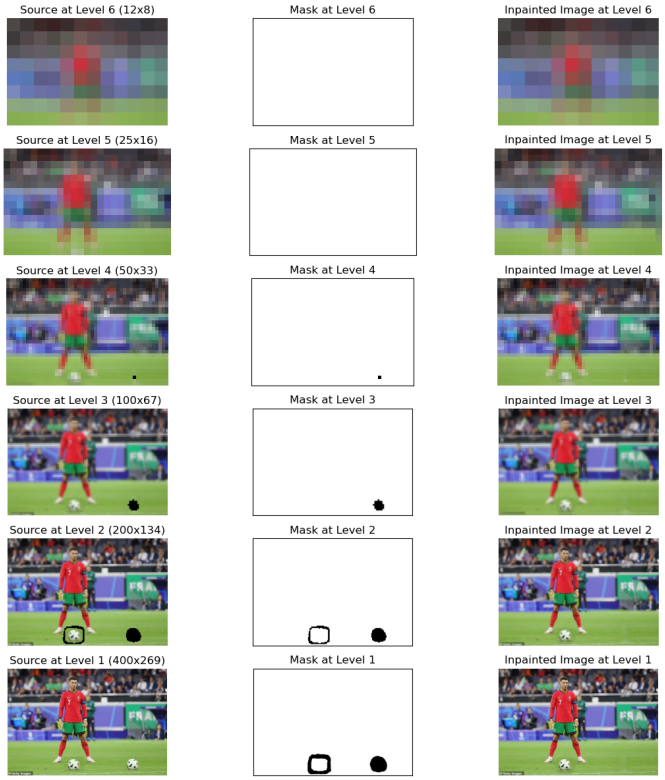

Fig. 12. Image Reshuffling using ronalado image

Hole Filling

Hole filling is a critical component of image reshuffling, as relocating an object leaves a void in its original position that must be plausibly reconstructed. Without this step, the edited image would contain an obvious artefact where the object was removed, resulting in a visually inconsistent or incomplete output.

In our framework, hole filling is performed using the same PatchMatch-based in-painting pipeline used for general reconstruction tasks. Once an object is copied and pasted to a new location, a binary mask is generated to mark the pixels corresponding to the object's original region. This mask guides the in-painting algorithm by specifying which pixels must be synthesised and which remain untouched.

The hole filling process operates in a multi-scale coarse-to-fine manner. At the coarsest pyramid level, the algorithm first establishes a rough structural estimate for the missing region by matching patches from the surrounding context. These initial estimates capture global appearance and layout. As the reconstruction proceeds to finer levels, PatchMatch refines the Nearest Neighbour Field, enabling the algorithm to synthesise local textures with higher fidelity. The Expectation–Maximisation (EM) updates further ensure that the filled region blends smoothly with the neighbouring pixels, both in colour and structural continuity.

This combination of patch-based matching, EM refinement, and multi-resolution optimisation results in a hole-filling process that produces natural, seamless completions. Even for large missing areas or complex textures, the algorithm effectively leverages the redundancy present in the image to generate visually coherent content. Consequently, hole filling becomes an essential step in maintaining realism and preserving the perceptual quality of the reshuffled image.

Hole Filling

Hole filling is an essential step in image reshuffling because moving an object leaves behind an empty region that must be visually completed. After the object is relocated, a binary mask marks the original region as missing, and PatchMatch-based in-painting is used to synthesise new content.

The hole is filled using a coarse-to-fine reconstruction pipeline: coarse levels provide a rough structural estimate, while finer levels refine texture details. Through patch propagation, random search, and EM-based updates, the algorithm generates content that blends naturally with the surrounding pixels. This ensures that the removed region is reconstructed seamlessly, producing a realistic reshuffled image without visible artefacts.

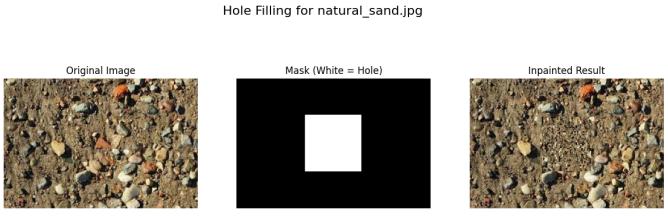

Fig. 13. Hole filling for the stones

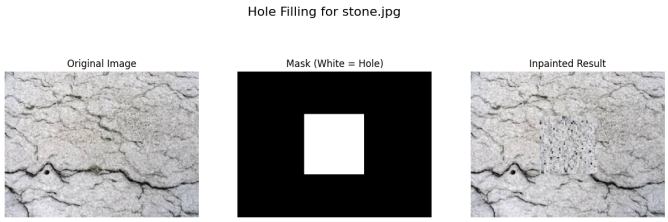

Fig. 14. Hole filling for the cracked road

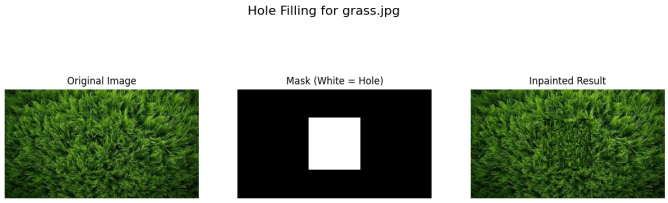

Fig. 15. Hole filling in the grass

Image Restoration

Image restoration focuses on recovering or enhancing the visual quality of an image that has been degraded due to noise, damage, blur, or missing regions. The objective is to reconstruct the image so that its underlying structure and appearance are preserved while correcting the imperfections introduced by degradation.

Using our in-painting framework, restoration is performed by identifying the corrupted or missing regions through a binary mask and reconstructing them using PatchMatch-based synthesis. We implemented and compared two distinct reconstruction strategies:

Best-Match Reconstruction

In the standard approach, each pixel in the reconstructed image is simply copied from the center of the best-matching patch found in the source image. While fast, this can sometimes lead to visible seams or blocky artifacts if the transition between adjacent patches is not perfectly smooth.

Voting-Based (Average) Reconstruction

To improve visual coherence, we implemented a voting-based reconstruction method. Since patches overlap significantly, a single pixel belongs to multiple overlapping patches (e.g., up to $7 \times 7 = 49$ patches for a patch width of 7). Instead of taking a single value, we aggregate the color values from all overlapping patches that cover a specific pixel and compute their average. This averaging process reduces blockiness, smooths transitions between neighbouring patches, and produces a more consistent reconstruction. By combining information from multiple overlapping matches, the voting-based method is able to compensate for local patch errors and reduce the impact of occasional mismatches. As a result, it generates restorations

that exhibit fewer seams and noticeably more natural texture continuity.

Fig. 16. Image Reconstruction

Fig. 17. Image Reconstruction Normal method and average

Fig. 18. Input Images

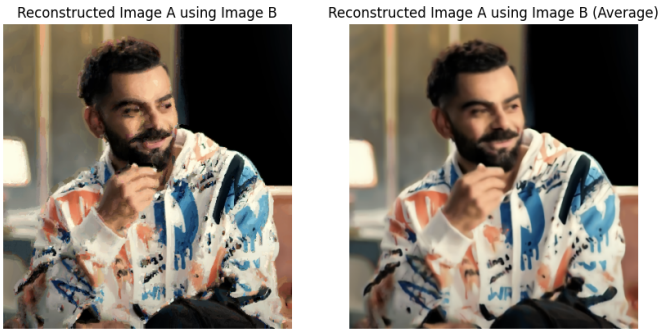

Fig. 19. Image Reconstruction Normal method and average

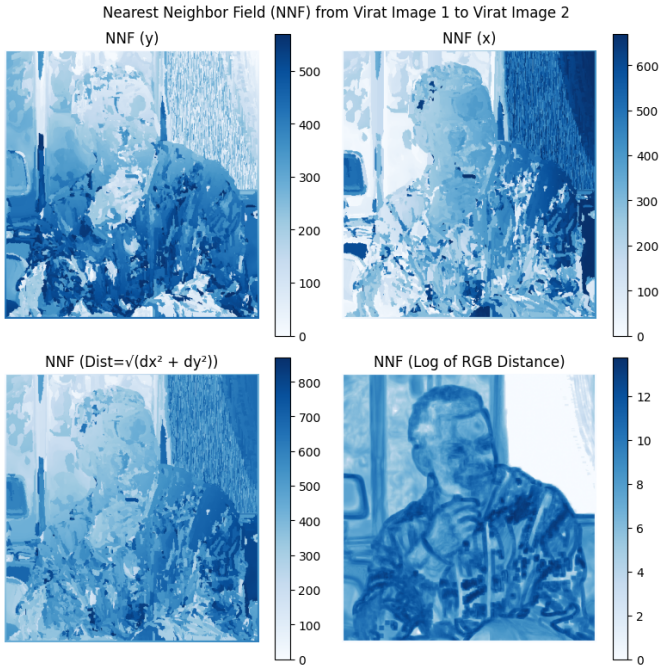

Fig. 20. NNF of virat kholi pictures

Fig. 21. Artistic Style Transfer

Fig. 22. Texture Swapping

Fig. 23. Creative Combination

Interactive Tool

To make the in-painting algorithm accessible and easy to use, we developed a Graphical User Interface (GUI) using Python's `tkinter` library. This tool abstracts the

command-line complexity and provides an intuitive workflow for image editing.

The GUI allows users to:

- **Load Images:** Import any standard image format for processing.
- **Draw Masks:** Interactively mark regions to be removed or in-painted using a brush tool. The interface includes controls to adjust the brush size (1 to 50 pixels) and an eraser mode to correct the mask selection.
- **Real-time Feedback:** Visualise the mask overlay on the original image before processing.
- **Save Results:** Export the final in-painted image to the disk.

For Demonstration:
https://drive.google.com/file/d/1q8J2ZfXef0MhYhAUJC7-dCw5oD0rQ8e2/view?usp=sharing

The tool integrates directly with our `InpaintNNF` class, passing the user-generated mask and image to the backend pipeline and displaying the result upon completion.

Performance Analysis

We conducted a theoretical and empirical analysis of the algorithm's performance to understand its computational characteristics.

Time Complexity

The time complexity of the PatchMatch algorithm is dominated by the patch distance calculations performed during the propagation and random search phases.

- **Per-Iteration Complexity:** For an image of height H and width W, and a patch width of p, the complexity is $O(H \cdot W \cdot p^2)$.
- **Total Complexity:** With k iterations, the total complexity is $O(k \cdot H \cdot W \cdot p^2)$.

Although we employ a multi-scale pyramid approach, the total number of pixels decreases geometrically at each coarser level (by a factor of 4). Consequently, the total work across all levels is bounded by a constant factor (approximately $\frac{4}{3}$) of the work at the finest resolution, preserving the overall complexity class.

Empirical Results

We measured the execution time across varying image resolutions and patch sizes. Our results confirm that the runtime scales linearly with the number of pixels and quadratically with the patch width, consistent with the theoretical derivation.

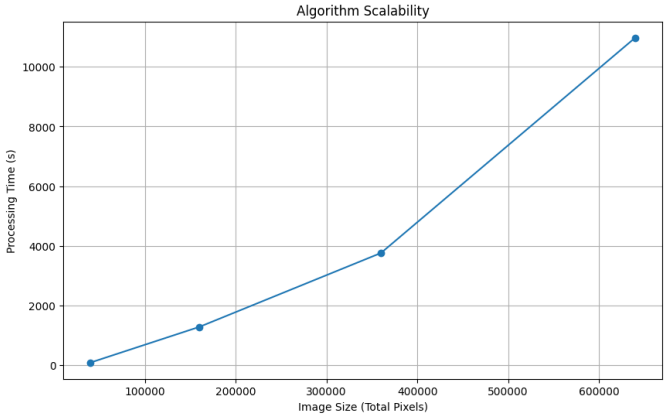

Fig. 24. Algorithm scalability

Parameter Sensitivity

To determine the optimal hyperparameters for the in-painting task, we conducted a parameter sensitivity study, specifically focusing on the impact of **Patch Size**.

We evaluated patch sizes of 3×3, 5×5, 7×7, 9×9, and 11×11 on a standard test set. For each size, we measured:

- **Reconstruction Quality:** Measured using Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM).
- **Computational Cost:** Measured as the total execution time.

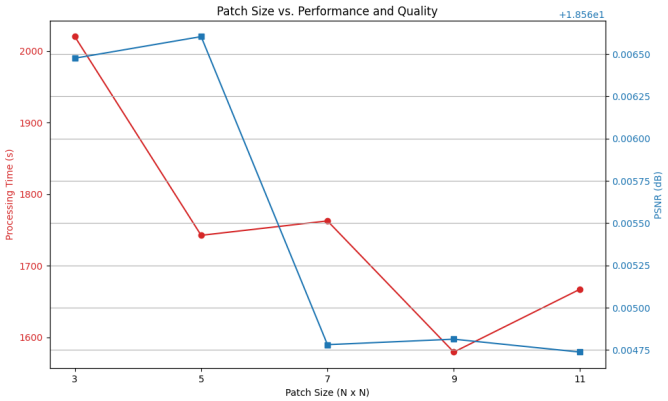

Fig. 25. Patch Size vs Performance and Quality

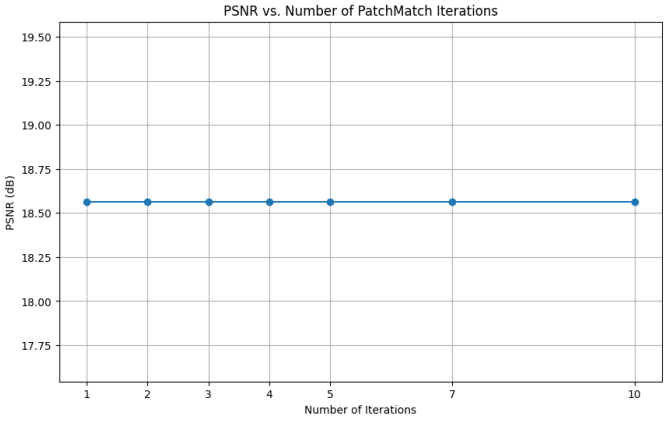

Fig. 26. PSNR vs Number of iterations

Observations: Smaller patches (3×3) capture fine details but often fail to reconstruct larger structural features, leading to "garbage" texture synthesis in large gaps. Larger patches (11×11) preserve structure better but significantly increase computational time and can result in over-smoothing. We found that a patch size of 7×7 offers the best trade-off between structural coherence, texture detail, and processing speed.

Comparative Analysis

To quantitatively evaluate the effectiveness of our implementation, we compared the PatchMatch results against baseline methods using standard image quality metrics.

Evaluation Metrics

We implemented the following metrics in `metrics.py` to assess reconstruction quality:

- **PSNR (Peak Signal-to-Noise Ratio):** Measures the ratio between the maximum possible power of a signal and the power of corrupting noise. Higher values indicate better reconstruction.
- **SSIM (Structural Similarity Index):** A perceptual metric that quantifies the degradation in structural information. Values range from -1 to 1, with 1 indicating perfect similarity.
- **MSE (Mean Squared Error):** Measures the average squared difference between the estimated values and the actual value.

Quantitative Comparison Across Test Images

Test Image	PSNR (dB)	SSIM	Time (s)
watermark	24.77	0.8752	269.72
football_sky	18.56	0.9419	1874.64
photo_bomb	35.93	0.9815	216.17
balls	25.26	0.9750	121.38
tattoo	17.67	0.9053	145.72

Fig. 27. Quantative Comparsion across some test images

Comparison Results

Our analysis shows that the PatchMatch-based in-painting significantly outperforms simple diffusion-based techniques, particularly in recovering textured regions. While Deep Learning approaches may offer superior semantic understanding for complex scenes, PatchMatch remains a highly efficient and effective solution for texture synthesis and object removal without requiring extensive training data.

References

[1] C. Barnes, E. Shechtman, A. Finkelstein, and D. Goldman, "PatchMatch: A randomized correspondence algorithm for structural image editing," *ACM Transactions on Graphics (SIGGRAPH)*, vol. 28, no. 3, pp. 24:1–24:11, 2009. Available: https://dl.acm.org/doi/pdf/10.1145/1531326.1531330

[2] C. Barnes, "Patch-based Approximate Nearest Neighbour Algorithms for Image Editing Applications," Ph.D. dissertation, Princeton University, 2011. Available: https://gfx.cs.princeton.edu/pubs/_2011_PAF/connelly_barnes_phd_thesis.pdf

[3] Z. Pei, "PatchMatch-based Image Inpainting," GitHub repository, 2018. Available: https://github.com/ZQPei/patchmatch_inpainting

[4] The Independent Code, "PatchMatch Algorithm Explained — Visual and Intuitive Guide," YouTube video, 2020. Available: https://www.youtube.com/watch?v=fMe19oTz6vk