Industrial Oriented Mini Project Report
On

ADVANCED IMAGE INPAINTING TECHNIQUES FOR IMPROVED SCENE ANALYSIS

Submitted to Jawaharlal Nehru Technological University for the partial Fulfillment of the Requirement for the Award of the Degree of

Bachelor of Technology
In
Computer Science and Engineering

By

G.KALYAN (22RA1A0547)
B.ASHWINI (22RA1A0519)
B.KESHAV (22RA1A0552)

Under the guidance of
Mrs . R. NIRANJANI
M.Tech
Assistant Professor
Dept. of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY

(UCG AUTONOMOUS)

(Affiliated to JNTUH, Ghanpur (V), Ghatkesar (M), Medchal (D)-500088)

2022-2026

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY
(UGC AUTONOMOUS)

(Ghanpur (V), Ghatkesar (M), Medchal (D)-500088)

(AFFILIATED TO JNTU, Hyderabad)

CERTIFICATE

This is to certify that the Mini Project entitled **" ADVANCED IMAGE INPAINTING TECHNIQUES FOR IMPROVED SCENE ANALYSIS"** being submitted by **G. KALYAN (22RA1A0547), B. ASHWINI (22RA1A0519), B. KESHAV (22RA1A0552)** in partial fulfillment for the award of Bachelor of Technology in Computer Science and Engineering to the Kommuri Pratap Reddy Institute Of Technology is a record of confined work carried out by them under my guidance and supervision.

Guide's signature
Mrs.R.NIRANJANI
M.Tech
Asst. Professor
Dept. of CSE

Signature of the Head of the Dept.
K.VAMSHEE KRISHNA
M.Tech, (Ph.D).
Asst. Professor
Dept. of CSE

Place: Ghanpur

Date:

Signature of External Examiner

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY

(UCG AUTONOMOUS)

(Ghanpur (V), Ghatkesar (M), Medchal (D)-500088)

(AFFILIATED TO JNTU, Hyderabad)

DECLARATION

We, **G. KALYAN (22RA1A0547), B. ASHWINI (22RA1A0519), B. KESHAV (22RA1A0552)** hereby declare that the mini project report titled **"IMAGE INPAINTING"** under the guidance of **Mrs. R. NIRANJANI**, Assistant Professor, Department of Computer Science and Engineering, Kommuri Pratap Reddy Institute of Technology, Ghanpur, is submitted in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering.

This is a record of bonafide work carried out by us and the results embodied in this project have not been reproduced or copied from any source. The results embodied in this thesis have not been submitted to any other University and Institute for the award of any Degree or Diploma.

G. KALYAN (22RA1A0547)

B. ASHWINI (22RA1A0519)

B. KESHAV (22RA1A0552)

ACKNOWLEDGEMENT

We manifest our heartier thankfulness pertaining to our contentment over **Mrs. R. NIRANJANI**, Assistant Professor as a project guide with whose adroit concomitance the excellence has been exemplified in bringing out this project work with artistry. We express our gratitude to **Mr. K. Vamshee Krishna**, Head of the Department for the encouragement and assistance to us, which contribute to the successful completion of this project. This Acknowledgement will be incomplete without mentioning our sincere gratefulness to our honorable Chairman **Sri. Kommuri Pratap Reddy**, our Director, **Dr. B. Sudheer Prem Kumar,** and Vice Principal, Dr.Sreenath Kashyap, who have been observing posing valiance in abundance, forwarding our individuality to acknowledge our project work tendentiously. At the outset we thank teaching and non-teaching staff of Dept. of CSE for providing us with good faculty and for their moral support throughout the course.

A heartful and sincere gratitude to our beloved parents for their tremendous motivation and moral support. We also express the overall exhilaration and gratitude to all those who animated our project work and accentuated our stance.

G. KALYAN (22RA1A0547)

B. ASHWINI (22RA1A0519)

B. KESHAV (22RA1A0552)

KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Vision of the Institute

To emerge as a premier institute for high quality professional graduates who can contribute to economic and social developments of the Nation.

Mission of the Institute

Mission	Statement
IM1	To have holistic approach in curriculum and pedagogy through industry interface to meet the needs of Global Competency.
IM2	To develop students with knowledge, attitude, employability skills, entrepreneurship, research potential and professionally Ethical citizens.
IM3	To contribute to advancement of Engineering & Technology that would help to satisfy the societal needs.
IM4	To preserve, promote cultural heritage, humanistic values and Spiritual values thus helping in peace and harmony in the society.

KOMMURI PRATAP REDDY INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Vision of the Department

To Provide Quality Education in Computer Science for the innovative professionals to work for the development of the nation.

Mission of the Department

Mission	Statement
DM1	Laying the path for rich skills in Computer Science through the basic knowledge of mathematics and fundamentals of engineering
DM2	Provide latest tools and technology to the students as a part of learning infrastructure
DM3	Training the students towards employability and entrepreneurship to meet the societal needs.
DM4	Grooming the students with professional and social ethics.

Program Educational Objectives (PEOs)

PEO's	Statement
PEO1	The graduates of Computer Science and Engineering will have successful career in technology.
PEO2	The graduates of the program will have solid technical and professional foundation to continue higher studies.
PEO3	The graduate of the program will have skills to develop products, offer services and innovation.
PEO4	The graduates of the program will have fundamental awareness of industry process, tools and technologies.

Program Outcomes

PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, Engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem Analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental context, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9	Individual and team network: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, being able to comprehend and write effective reports and design documentation, make Effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environment.
PO12	Life-Long learning: Recognize the need for, and have the preparation and able to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

PSO1	Foundation of mathematical concepts: To use mathematical methodologies to crack problem using suitable mathematical analysis, data structure and suitable algorithm.
PSO2	Foundation of Computer Science: The ability to interpret the fundamental concepts and methodology of computer systems. Students can understand the functionality of hardware and software aspects of computer systems.
PSO3	Foundation of Software development: The ability to grasp the software development lifecycle and methodologies of software systems. Possess competent skills and knowledge of software design process.

ABSTRACT

This work presents advanced image inpainting techniques aimed at improving scene analysis by restoring missing regions of images in a way that maintains contextual integrity and enhances visual coherence. Image inpainting is a key technique in image processing, particularly useful for applications where image restoration or object removal is required. Traditional methods, such as nearest neighborhood-based pixel replacement, offer basic solutions but struggle with large or complex missing regions. These limitations have led to the development of the Iterative Fast March Inpainting (IFM) method, which offers a more sophisticated and iterative approach to inpainting, producing superior results in handling complex and irregularly shaped missing regions. The proposed Iterative Fast March Inpainting method works by progressively filling in missing regions based on the surrounding image context, utilizing a Fast Marching Method that respects image gradients and structure. This iterative approach ensures that the restoration process is both accurate and contextually aware, leading to more natural and realistic inpainting results. The technique is particularly effective in scenarios where traditional methods fail, such as large-scale inpainting or complex object restoration, providing a powerful tool for improving the quality of scene analysis in various applications. Through extensive performance evaluations, the study demonstrates that the IFM method outperforms existing approaches in terms of restoration quality, computational efficiency, and flexibility.. By offering a robust solution for large-scale inpainting challenges, IFM significantly enhances the capabilities of scene analysis and image restoration.

Keywords:

- *Image Inpainting*
- *Traditional Methods*
- *Scene Analysis*
- *Patch-based Techniques*
- *Diffusion-based Models*

INDEX

DIAGRAMS

TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Image inpainting, the process of reconstructing lost or deteriorated parts of images, has evolved from traditional manual restoration to advanced deep learning-based methods. The global image inpainting market is expected to grow significantly, fueled by its rising integration in various domains such as surveillance, medical imaging, satellite imaging, and digital art restoration [1,2]. According to Allied Market Research, the global AI image analysis market was valued at $1.4 billion in 2020 and is projected to reach $13 billion by 2030, growing at a CAGR of 25.3%. A significant portion of this growth is driven by the increasing demand for automated image reconstruction and correction, especially in environments with missing or corrupted visual data [3].

Scene analysis, the interpretation of a visual environment through computational models, requires consistent and complete input. Inpainting enhances the completeness of images by filling in missing regions, thereby boosting scene interpretation algorithms. For example, in autonomous driving systems, occluded regions of a street view—such as objects hidden behind poles or trees—can be reconstructed for better decision-making. Similarly, in video surveillance, damaged frames due to transmission loss or environmental interference can be corrected using advanced inpainting models to maintain continuity and accuracy in behavior analysis. A study by NVIDIA in 2021 demonstrated that image inpainting boosted object detection precision by up to 14% in partially corrupted frames [4, 5].

Over the years, the methods have shifted from exemplar-based models and patch-based synthesis to more efficient deep generative approaches. Convolutional neural networks (CNNs), generative adversarial networks (GANs), and transformers have revolutionized the capabilities of image inpainting, making it context-aware and semantically consistent. These models do not just focus on pixel-level completion but also ensure structural integrity and perceptual realism. The quality of restored content is crucial in scenarios like heritage restoration and remote sensing, where even minor inaccuracies can lead to misinterpretation. The performance of modern inpainting techniques is often measured using PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index), and FID (Fréchet Inception Distance), with values consistently improving across recent studies.

1.2 Research Motivation

Real-time applications such as autonomous vehicles, medical diagnostics, and satellite imaging face a common challenge—missing or damaged visual data. Tesla and Waymo, pioneers in self-driving technology, rely heavily on complete visual scene interpretation for navigation and obstacle avoidance. Any missing or obstructed portion of a street scene, like a pedestrian obscured by a parked vehicle, can pose a risk. Inpainting techniques help reconstruct those occluded areas, allowing the system to maintain a coherent understanding of its environment. This significantly improves the decision-making process and enhances the safety and efficiency of autonomous systems.

In the healthcare industry, companies such as Siemens Healthineers and GE Healthcare utilize advanced image analysis for diagnostics using MRI, CT scans, and X-rays. Often, due to noise, motion artifacts, or hardware limitations, parts of the medical images may be unclear or missing. Inpainting algorithms assist in reconstructing these portions, allowing radiologists and AI diagnostic tools to get a complete and reliable view. This is critical for detecting anomalies like tumors or fractures that might otherwise remain hidden. A complete and reliable visual dataset ensures accurate diagnosis, ultimately improving patient outcomes and reducing the chances of false positives or negatives.

In satellite imaging and environmental monitoring, organizations like NASA and ESA depend on high-quality, continuous imagery for tasks like deforestation analysis, climate monitoring, and urban planning. Satellite images often suffer from cloud occlusion or data loss due to transmission errors. Inpainting helps fill these gaps, providing analysts with uninterrupted and clean visual data. This allows for better forecasting, planning, and emergency response. Moreover, real-time inpainting capabilities can drastically reduce data processing delays in mission-critical systems such as disaster response and military surveillance.

1.3 Problem Definition

Despite rapid advancements in computer vision, image inpainting still faces several critical challenges that limit its effectiveness in real-world applications. Most traditional and early deep learning approaches struggle with large missing regions, particularly when the occlusion covers semantically important areas like faces, text, or structural elements. This results in unrealistic or blurry reconstructions that can hinder further scene analysis or automated interpretation.

Additionally, ensuring context-aware and perceptually consistent reconstruction remains a significant hurdle. Many models fail to understand high-level semantics, leading to mismatches in texture, structure, or lighting between the reconstructed and original parts of an image. This inconsistency is problematic in professional applications such as heritage restoration, forensic investigation, and urban planning where visual accuracy is critical. Moreover, training inpainting models requires vast datasets and computational resources, which may not be feasible for all developers or institutions.

Furthermore, the lack of generalizability across domains—such as transferring an inpainting model trained on natural scenes to perform effectively on medical or satellite imagery—is another major limitation. Models that perform well in one domain may not adapt to another due to variations in texture, resolution, and structural complexity. This presents an ongoing need for more robust, domain-adaptive inpainting solutions that can operate reliably in diverse real-time scenarios.

1.4 Significance

Image inpainting is more than a restoration tool; it is a foundational element for multiple advanced vision-based systems. Its ability to recover lost visual information directly influences the performance of higher-level tasks such as object detection, tracking, and semantic segmentation. Enhancing the completeness of input images reduces error rates in AI systems, ensuring more reliable automation and analysis.

As industries increasingly lean toward data-driven decisions, the quality and integrity of visual input become vital. Inpainting provides a way to address incomplete data without manual intervention, making systems more autonomous and less prone to failure. This holds especially true in applications that operate in dynamic and unpredictable environments, such as drones, security systems, and medical robots.

Moreover, inpainting contributes to reducing data wastage. Instead of discarding images with corrupted or missing regions, these can be reconstructed and utilized effectively, maximizing the use of available data. This is particularly important in costly or limited-data fields like space exploration and bio-imaging. Thus, advanced inpainting is not just about aesthetics—it's about functionality, efficiency, and intelligent decision-making support.

1.5 Proposed Objectives

The proposed Iterative Fast March Inpainting (IFM Inpainting) method aims to overcome the limitations of nearest neighborhood-based approaches by using an iterative algorithm to progressively restore missing image regions. This technique is based on the Fast Marching Method (FMM), which is typically used for solving boundary value problems in numerical simulations. In the context of image inpainting, IFM operates by propagating information from the known regions of the image into the unknown regions in a way that respects the underlying structure and gradients of the image. The iterative process allows for better handling of large or irregularly shaped missing areas by refining the inpainted regions in each iteration.

The iterative nature of the Fast Marching Inpainting method ensures that the inpainting process is more context-aware, as it takes into account both local and global image information. The algorithm iteratively fills in the missing regions by analyzing the gradients and boundaries of surrounding areas, ensuring that the restored regions blend seamlessly with their surroundings. This method is particularly effective for handling complex images with large areas of missing data, as it can generate high-quality, realistic inpainting results even in challenging scenarios. Additionally, the iterative process allows for fine-tuning the inpainted areas, improving the overall accuracy and coherence of the final result.

Advantages of Iterative Fast March Inpainting

The Iterative Fast March Inpainting method offers several advantages over traditional image inpainting techniques. One of its key strengths is its ability to handle large, irregularly shaped regions of missing data more effectively than nearest neighborhood-based methods. By leveraging the Fast Marching Method, IFM ensures that the inpainting process respects the underlying image structure and gradients, producing results that are visually coherent and contextually accurate. This method also excels in maintaining global image consistency, particularly in scenarios where the missing regions are part of more complex objects or textures.

Another significant advantage of IFM is its iterative nature, which allows for progressive refinement of the inpainting results. The algorithm can adjust and improve the restored areas with each iteration, leading to more accurate and natural inpainting in challenging scenarios. This iterative process ensures that the inpainted regions gradually converge to a more realistic

representation of the missing parts, resulting in higher-quality restorations. Moreover, IFM is flexible and can be adapted for various types of images, including natural scenes, portraits, and even images with high-frequency details. Its ability to handle diverse inpainting challenges makes it a robust solution for advanced image inpainting tasks.

1.6 Advantages

- Enhances the accuracy of object recognition and classification in partially corrupted or occluded images.
- Reduces the need for data rejection, increasing dataset utilization and efficiency.
- Improves the performance of downstream tasks such as tracking, segmentation, and scene understanding.
- Enables real-time visual correction in surveillance and autonomous vehicle systems.
- Supports restoration of historical and degraded artwork or photographs with minimal manual input.
- Provides robust reconstruction in medical diagnostics, aiding in more precise interpretations.
- Enhances consumer experiences in AR/VR applications by filling in occluded virtual scenes.
- Reduces human effort and cost associated with manual editing or correction in multimedia content.
- Contributes to cleaner satellite imagery for environmental monitoring and planning.
- Facilitates efficient forensic analysis by recovering visually obscured or tampered regions.

1.7 Applications

- Autonomous driving systems utilize inpainting to reconstruct occluded objects for improved navigation.
- Medical imaging platforms apply inpainting to restore noisy or incomplete MRI/CT scans.

- Satellite and aerial imaging tools use inpainting for cloud cover removal and terrain reconstruction.
- Augmented and Virtual Reality systems benefit from scene completion for immersive experiences.
- Surveillance systems employ inpainting to repair corrupted video frames in real-time monitoring.
- Digital content creators use it in photo/video editing software to remove unwanted objects.
- Heritage conservation projects use inpainting to restore damaged historical artworks and documents.
- Underwater robotics apply inpainting to reconstruct obscured images caused by murky waters.
- Fashion and e-commerce platforms use it to generate complete views of products from limited imagery.
- Space exploration missions use inpainting to fill gaps in data collected from distant planetary surfaces.

CHAPTER 2

LITERATURE SURVEY

Image inpainting is sometimes called an inverse problem, and usually these types of problems are ill-posed. The problem of inpainting consists in finding the best approximation to fill in the region inside the source image and comparing it with the ground truth. All the algorithms that tackle this problem begin with the assumption that there must be some correlation between the pixels present inside the image, either from a statistical or from a geometrical perspective. The objective of inpainting is to minimize the difference between the original image, I, and the reconstructed image, R, in the domain, D. This is typically achieved by defining an appropriate loss function that quantifies the differences between I and R in region D. In other words, the inpainting problem can be formulated as an optimization problem that minimizes an objective function. The objective function typically consists of two main components: a term that penalizes the deviation of the inpainted values from the known values in the known region, D; and a term that penalizes large variations in the inpainted values, encouraging smoothness and preventing overfitting. Thus, the first operation ensures that the inpainted image is consistent with the available information, and the second helps to preserve the natural structure and appearance of the original image. The inpainting methods can be split into the following three categories:

- Diffusion-based or sometimes called partial differential equations-based (here, we are also going to include TV methods as well);
- Exemplar-based or patch-based, as referred to in some other papers;
- Machine learning is undertaken irrespective of their model architecture.

The following section focuses on describing the most cited and recent state-of-the-art methods to better understand, the overall artifacts introduced in the inpainting procedure.

2.1. Diffusion-Based Methods

The term diffusion (from a chemistry point of view) is an action in which items inside a region of higher concentration tend to move to a lower concentration area. In the analysis undertaken in [6], in which the inpainting process is inspired by the “real” inpainting of canvas, the process consists of the following steps. The first step in all the inpainting

algorithms is to apply some sort of regularization. It can be isotropic, with some poor results, anisotropic, or any other type of regularization. This is performed to ensure that image noise is removed, so that it shall not interfere in the computation of the structural data needed in the next step.

To apply diffusion, the structural and statistical data of the low-level image must be identified. Based on this data, if on an edge in the δD area, the algorithm must conserve the edge identified, and if the δD area belongs to a consistent area, the algorithm can then easily replicate the same pixel information from the border. To retrieve image geometry, one can use isophotes, which are curved on-surface connecting points of the same values. For this, one needs to first compute the gradient on each point in the margin area and then compute the direction as normal in relation to the discretized gradient vector.

Having performed these steps, the initial algorithm from [6] is just a succession of anisotropic filtering, followed by inpainting, and then, this is repeated several times. Later, the authors in [7] proposed an improved version of their initial algorithm. This idea was inspired from the mathematical equations of fluid dynamics, specifically the Navier–Stokes equations, which describe the motion of fluid. The proposal was to use the continuity and momentum equations of fluid dynamics to propagate information from known areas of the image towards the missing or corrupted areas. This was an improved version of the higher PDE version presented initially. As a follow-up of his original work, Bertalmio proposed the use of third-order PDE in [8], which is a better continuation of edges. At the same time, Chan and Shen developed similar algorithms [9,10], in which they postulated the use of the local curvature of an image to guide the reconstruction of missing or obscured parts. Using Euler's elastica model, they could predict what the missing parts of an image might look like. Both Euler's elastica and PDE-based inpainting are effective methods for image inpainting and have their own advantages and disadvantages. Euler's elastica is particularly well-suited for images that contain thin, flexible objects, while PDE-based inpainting is well-suited for images that are smooth and locally consistent. Depending on the specific characteristics of the image and the desired outcome, one method may be more appropriate than the other. In recent year, the focus for diffusion-based inpainting has moved towards increasingly complex PDE forms; e.g., in [11] using high-order variational models is suggested, like low curvature image simplifiers or the Cahn–Hilliard equation. Another recent paper that goes in the same direction is [12], which integrates the geometric features of an image—namely the Gauss curvature. Still, even these methods introduce the blurring artifact also found in the initial

papers [6,7]. To surpass these challenges in the current models, with second-order diffusion-based models that are prone to staircase effects and connectivity issues and fourth-order models that tend to exhibit speckle artifacts, a newer set of models must be developed. The authors Sridevi and Srinivas Kumar proposed several robust image inpainting models that employ fractional-order nonlinear diffusion, steered by difference curvature in their papers [13,14,15]. In their most recent paper [16], a fractional-order variational model was added to mitigate noise and blur effectively. A variation of DFT is used to consider pixel values from the whole image, and this is not by relying strictly on only the neighboring pixels.

Another method that yields better results than standard PDE is the use of total variation inpainting. The core idea is to minimize the total variation of the inpainted image, effectively reducing the abrupt changes in intensity. TV inpainting incorporates a regularization term into the optimization problem. This term typically comes in two types: L1 and L2 regularization. L1 regularization encourages sparsity in the gradients of the inpainted image, promoting piecewise constant solutions with sharp edges. On the other hand, L2 regularization results in smoother images with gradual transitions. The choice between these regularization terms determines the trade-off between smoothness and fidelity in the inpainted result. First-order TV inpainting focuses on minimizing the L1-norm of image gradients and is effective for restoring image edges and preserving fine structures. Minimizing the L1-norm of gradients encourages sparsity in the gradient domain, leading to piecewise constant solutions. One of the key advantages of first-order TV inpainting is its applicability in various scenarios, including image denoising and deblurring. This method excels in reconstructing images with well-defined edges and minimal noise. Second-order TV inpainting extends the TV concept by considering gradients of gradients, also known as the Laplacian operator. This extension allows for the preservation of more complex image structures, including textures and patterns. By considering higher-order derivatives, second-order TV inpainting can produce inpainted images with improved fidelity. However, second-order TV inpainting introduces computational challenges due to the increased complexity of the optimization problem such as in [17] where the split Bregman technique is used.

Based on the multitude of research on inpainting using PDE or TV methods, there are still some areas to be improved, and some artifacts are easily identifiable. In the following section, two methods are analyzed in terms of the artifacts introduced in the inpainting process. The first is based on the Cahn–Hilliard equation [11] and is a fourth-order PDE that describes the phase separation process in materials and has been adapted for inpainting tasks in image

processing, while the second analyzed method is based on total variation [17] and uses a combined first- and second-order total variation. The first analyzed sample is presented in Figure 3: on the left side is the original image [11], and on the right is the mask applied to that image. The mask created for the PDE method is created specifically for this test, and it tries to simulate a scratch on the image. The inpainting methods will try to generate pixels that are as accurate as possible to fill in the white line.

In the above scenario, the TV variation [17] is able to correctly reconstruct pixels, at least from a human perspective. However, upon examining the pixel values with the area values, it becomes apparent that the reconstruction is not flawless, as there are minor discrepancies in colors. The second analyzed scenario is more complex, and we are going to focus on the TV method only because the classic PDE method usually struggles with larger/highly texturized areas. In Figure 7, on the left side is the original image [17], and on the right is the mask applied to that image. The inpainting methods will try to generate pixels that are as accurate as possible to fill in the white line. In the below image, the result of the TV inpainting method can be observed.

2.2. Exemplar-Based Methods

At the same time, a newer approach based on texture synthesis started to gain more momentum. The main inspiration came from [18] in which A. A. Efros and T. K. Leung introduced a non-parametric method for texture synthesis, where the algorithm generates new texture images by sampling and matching pixels from a given input texture based on their neighborhood pixels, thus effectively synthesizing textures that closely resemble the input sample. This approach was notable for its simplicity and ability to produce high-quality results, making it a foundational work in texture synthesis. The primary goal of this approach was to enhance the reconstruction of the image section area that is missing. However, the challenges brought by texture synthesis are slightly different from those presented by classic image inpainting. The fundamental objective of texture synthesis is to generate a larger texture that closely resembles a given sample in terms of visual appearance. This challenge is also commonly referred to as sample-based texture synthesis. A considerable amount of research has been conducted in the field of texture synthesis, employing strategies such as local region growing or holistic optimization. One of the main papers that gained a lot of attention was the work of [19]. In this paper, Criminisi presented a novel algorithm for the removal of large objects from digital images. This technique is known as exemplar-based

image inpainting. This method is based on the idea of priority computation for the fill front, and the use of the best exemplar selection for texture synthesis. Given a target region, Ω, to be inpainted, the algorithm determines the fill order based on the priority function P(p), defined for each pixel, p, on the fill front: $\partial\Omega$. P(p) = C(p) * D(p), where C(p) is the confidence term, an indication of the amount of reliable information around pixel p; D(p) is the data term, a measure of the strength of the isophotes hitting the front at p. The algorithm proceeds in a greedy manner, filling in the region of highest priority first with the best match from the source region, Φ. This is identified using the sum of squared differences (SSD) between patches. The novel aspect of this method is that it combines structure propagation and texture synthesis into one framework, aiming to preserve the structure of the image while simultaneously considering the texture. It has been demonstrated to outperform traditional texture synthesis methods in many complex scenes, and it has been influential in the field of image processing.

In recent years, the methods have become increasingly complex and try to exploit various artifacts inside images and analyze more in-depth the structure near the area to be inpainted. Approaches like [20] utilize a patch-based approach that searches for well-matched patches in the texture component using a Markov random field (MRF). Jin and Ye [21] proposed an alternative patch-based method that incorporates an annihilation property filter and a low-rank structured matrix. Their approach aims to remove an object from an image by selecting the target object and restricting the search process to the surrounding background. Additionally, Kawai [22] presented an approach for object removal in images by employing a target object selection technique and confining the search area to the background. Authors have also explored patch-based methods for recovering corrupted blocks in images using two-stage low-rank approximation [23] and gradient-based low-rank approximation [24]. Another sub-area of focus for some authors was representing information by first “translating” the image into another format, the so-called sparse representation like DCT, DFT, DWT, etc. It is relevant to mention a few interesting research papers [25,26]. They obtained excellent quality while maintaining the uniformity of the area to be inpainted, but if the area is at the edge of various textures, the methods introduce some pretty ugly artifacts that make the methods unusable.

CHAPTER 3

EXISTING SYSTEM

3.1 Nearest Neighbourhood Based Pixel Replacement

Nearest Neighbourhood Based Pixel Replacement (NNBPR) for image inpainting is an image restoration technique used to fill in missing or corrupted parts of an image by utilizing information from the surrounding pixels. Here's a detailed, stepwise operational procedure for implementing NNBPR:

Step 1: Input Image: The first step is to acquire the image that needs inpainting. This image may contain missing regions or corrupted pixels, typically represented by a predefined marker (e.g., black pixels or NaN values).

Step 2: Mask Generation: A mask is created to identify the missing or corrupted parts of the image. The mask is usually a binary image where the missing pixels are marked with a value of 1 (indicating the pixel to be inpainted), and the rest are marked with 0 (indicating the intact pixels).

Step 3: Select the Region to Inpaint

- **Identify Missing Pixels**: Based on the mask generated in the preprocessing step, the missing or corrupted pixels are located. These pixels are the ones that will be replaced during the inpainting process.
- **Target Region for Replacement**: The missing pixels or the regions to be inpainted are segregated from the rest of the image using the mask.

Step 4: Nearest Neighborhood Search

- **Neighborhood Search Algorithm**: For each missing pixel, the algorithm searches for its nearest neighbors within the known (intact) part of the image. This can be done using various distance metrics such as Euclidean distance or Manhattan distance.
- **Selection of Nearest Neighbors**: The pixels surrounding the missing pixel in the intact region of the image are considered. The closest neighbors, which can be defined

in terms of spatial proximity or pixel color similarity, are selected for further consideration.

Distance Calculation: The distance between the missing pixel and its potential neighbors is calculated. For this step, the nearest neighborhood algorithm can use both spatial distance and intensity/color similarity metrics to ensure the replacement pixel is as close as possible in terms of both position and visual characteristics.

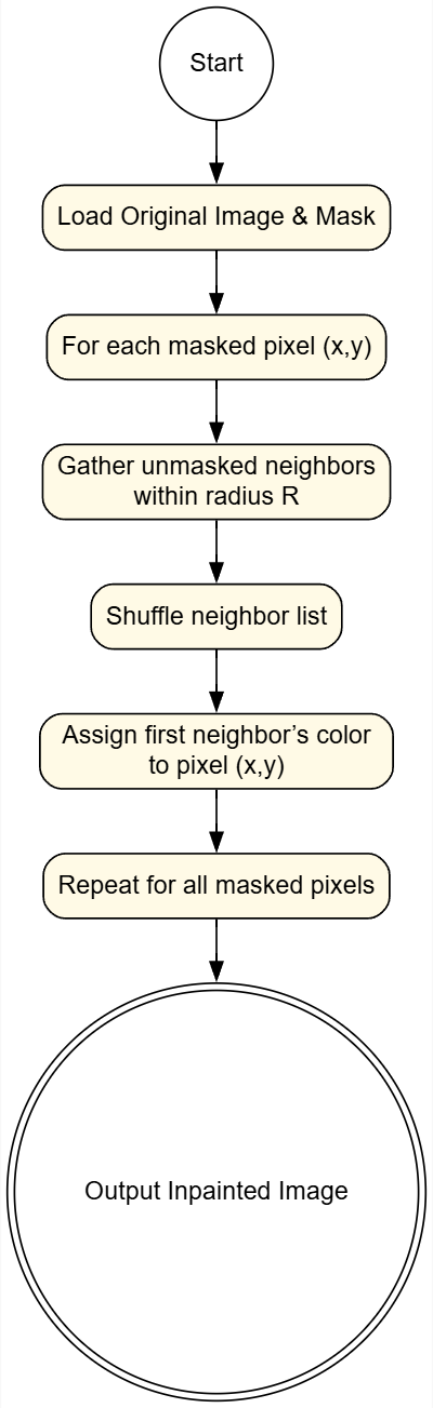


Figure 3.1. Existing NNBPRmethod.

Step 5: Pixel Replacement

- **Replacement Decision**: The replacement pixel value is determined by averaging or selecting the pixel value of the nearest neighbor(s) from the intact part of the image. Depending on the complexity of the algorithm, this could involve more sophisticated weighting based on similarity to preserve texture and edge consistency.
- **Iterative Inpainting**: For every missing pixel, this procedure is repeated. As pixels are filled in, the known region gradually expands, and neighboring pixels may begin to help fill subsequent missing pixels.
- **Handling Complex Regions**: In cases where a large area of the image is missing, the nearest neighbors may not always provide a good approximation of the inpainted regions. More advanced techniques like patch-based inpainting or using texture synthesis can be incorporated to improve the visual quality of the filled region.

3.2 Drawbacks

While the NNBPR method for image inpainting offers a straightforward and computationally efficient approach, it also has several drawbacks that can impact its performance in certain scenarios:

- **Limited Texture Representation**: NNBPR mainly relies on replacing missing pixels with values from the closest neighbors. In cases where the missing region contains fine details or complex textures, the algorithm may fail to capture the intricate patterns and textures present in the surrounding area, leading to unrealistic or blurred inpainting results.
- **Blurring of Edges**: The process of averaging or selecting the nearest neighbors for replacement can result in edge blurring. This is especially problematic when the missing region has sharp boundaries, as the inpainted pixels may lack the sharpness and definition needed to maintain visual consistency with the original image.
- **Challenges with Large Gaps**: NNBPR performs well for small or isolated missing regions, but its effectiveness significantly diminishes when large areas are missing. In such cases, the algorithm relies heavily on the nearest intact pixels, which might not provide enough information to accurately reconstruct the missing region. This can

lead to poor quality inpainting, especially in regions where the context is complex or where no similar pixels are available in the nearby intact area.

- **Reduced Global Context Awareness**: Since NNBPR is primarily a local method, it lacks the global context of the image, which makes it less effective when large portions of the image are missing, as it cannot incorporate distant regions' information for a more coherent inpainting.

CHAPTER 4

PROPOSED SYSTEM

4.1 Iterative Fast March Inpainting

The inpainting process is a structured approach to filling missing regions in an image by leveraging non-local similarity and iterative refinement. Figure 4.1 shows the proposed system architecture.

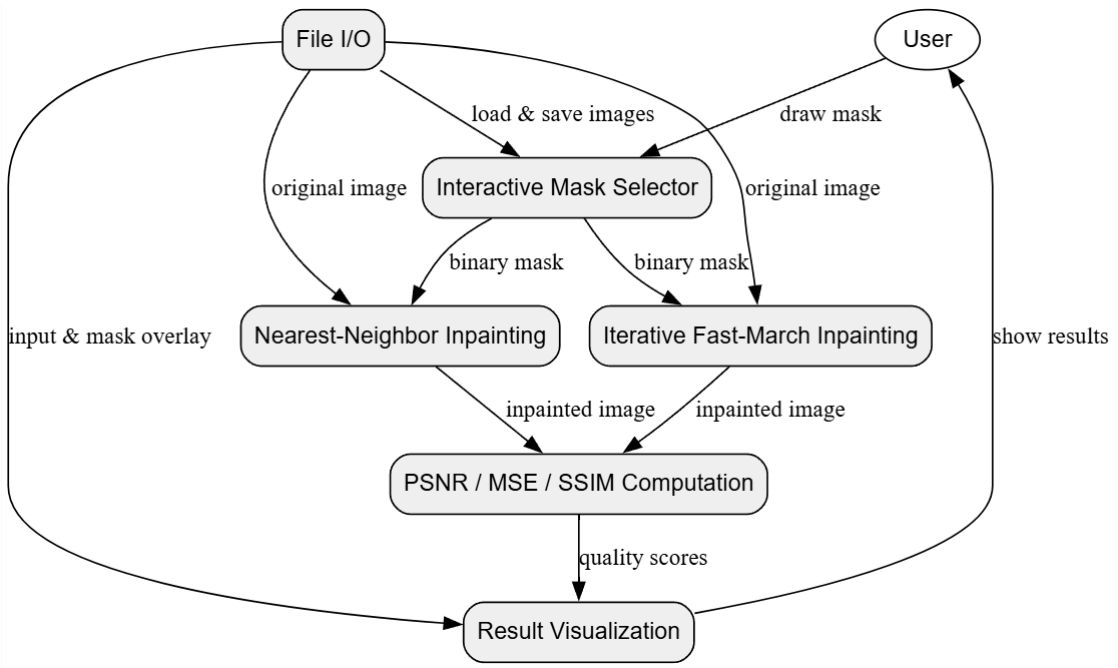


Figure 4.1. Proposed system architecture.

The detailed procedure is given as follows:

Step 1: Initializing the Inpainting Process

The inpainting process begins with the creation of an object that takes three key inputs: the original image, a mask image that identifies the missing regions, and a patch size value. The original image can be either grayscale or color, and it is provided as a file path. The mask image is a binary file where missing pixels are marked, typically with ones, while known pixels are represented by zeros. The patch size determines the extent of local neighborhoods that will be used to find replacement values for the missing regions. Once these inputs are provided, the object stores them as internal attributes and prepares to execute the inpainting procedure when called.

Step 2: Preparing the Image for Processing

Before any modifications are made to the image, it needs to be preprocessed. The first step in preprocessing is to load both the original image and the mask from their respective file paths. The pixel values of the original image are then normalized to ensure that they lie within a well-defined range, which prevents numerical issues during calculations. A small constant value is added to avoid division errors. Next, the mask is applied directly to the image, setting all missing pixels to zero while keeping the valid ones unchanged.

The image is then transformed into two feature representations: a position matrix and a texture matrix. The position matrix stores the spatial coordinates of each pixel, while the texture matrix holds the pixel intensity values. These two matrices are scaled so that their values remain within a comparable range. To further assist the inpainting process, the image is divided into smaller overlapping regions known as patches. These patches help in defining local neighborhoods for similarity comparisons and are generated using the specified patch size value.

Step 3: Identifying Missing and Known Pixels

Once the image is prepared, it is necessary to distinguish between pixels that need to be filled and those that are already known. This distinction is made using the texture matrix. The set of known pixels is determined by checking which locations contain valid intensity values, while the missing pixels are identified by looking for locations where all intensity values are zero. The missing pixels are stored separately as they will be the primary focus of the inpainting process. To efficiently search for neighboring pixels, a spatial data structure called a k-dimensional (k-d) tree is built from the position matrix, allowing for fast nearest-neighbor lookups.

A progress tracking system is initialized to monitor the number of missing pixels that remain to be inpainted. This system provides real-time updates as the process advances, ensuring that the procedure can be observed and adjusted if necessary.

Step 4: Iterative Filling of Missing Regions

The missing pixels are filled in a structured manner through an iterative process. Each iteration begins by identifying which of the missing pixels are located near the known pixels. This is done by selecting a small group of nearest neighbors for each missing pixel and

checking whether any of these neighbors are part of the known set. If a missing pixel has at least one known neighbor, it is marked as ready for inpainting.

Once a pixel is marked, its surrounding patch is extracted. A mask is applied to determine which parts of the patch contain valid information. A search is then conducted in the surrounding area to locate potential donor pixels, which are known pixels that can be used to estimate the missing value. These donor pixels are selected by searching within a predefined neighborhood radius. A temporary data structure is created to store the patches belonging to the donor pixels.

To determine the best estimate for the missing pixel, a similarity comparison is performed between the extracted patch and the donor patches. The closest matches are selected based on their similarity, and their intensity values are averaged to estimate the missing value. This estimated value is then assigned to the missing pixel.

Step 5: Updating the Image and Continuing the Process

After a group of missing pixels has been filled, the texture matrix is updated with the new values. The updated texture is then converted back into patches, ensuring that the process remains dynamic and responsive to changes. The set of missing pixels is adjusted by removing the ones that have been inpainted. The known pixel set is expanded to include these newly filled pixels, allowing for further iterations to be performed with an increasing number of reference points. The process repeats until all missing pixels have been assigned appropriate values.

The progress tracking system is continuously updated, providing real-time feedback on the number of remaining missing pixels. Each iteration reduces the number of missing pixels, gradually restoring the image to a complete form.

Step 6: Finalizing the Inpainting Result

Once all missing regions have been filled, the processed texture matrix is converted back into an image. The transformation ensures that the reconstructed image maintains the same dimensions as the original input. The final output is then returned as the completed inpainted image, ready for use.

This approach effectively restores missing parts of an image by leveraging a structured non-local search process. By carefully selecting appropriate donor pixels and iteratively refining

the missing regions, the inpainting process ensures a smooth and natural reconstruction of the original image.

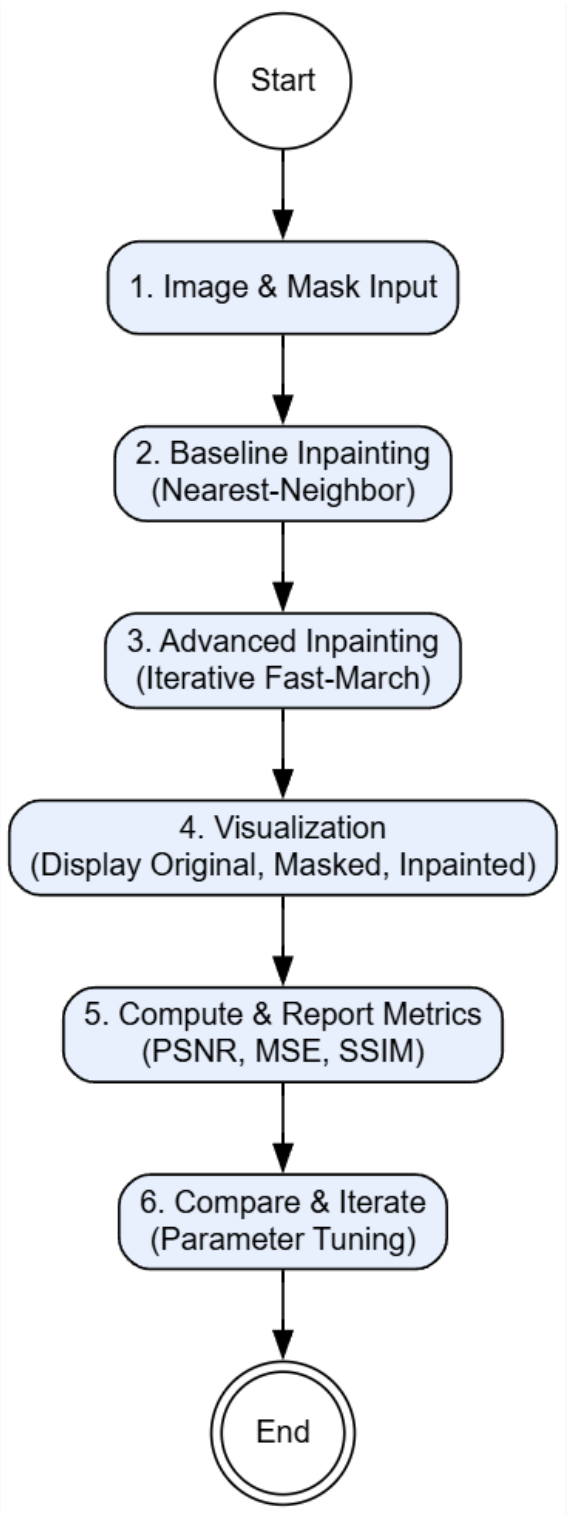


Fig. 4.2: Proposed workflow of image inpainting system.

4.2 Proposed workflow

The workflow of this inpainting project as demonstrated in Fig. 4.2 orchestrates interactive mask creation, two inpainting methods (baseline and Fast-March), and quantitative evaluation in a clear, repeatable pipeline. By modularizing each stage—input, processing, and output—you can easily extend or replace components (e.g., swap inpainting algorithms or metrics) without altering the overall structure. This organized flow ensures both visual and numeric comparisons are straightforward and makes it simple to batch-process multiple images or integrate a GUI for end-user interaction.

Workflow Steps

1. **Image & Mask Input**
    - **Load Original Image** (e.g. lincoln.png, Kid.jpg) via OpenCV.
    - **Obtain Binary Mask** either by:
        - Loading a pre-drawn mask image (*_mask.png), or
        - Launching the interactive mask drawer (select_regions) to freehand paint regions to remove.
2. **Baseline Inpainting: Nearest-Neighbor Replacement**
    - Iterate over each masked pixel.
    - Search within a ±5 px neighborhood for unmasked (white) pixels.
    - Randomly select one valid neighbor's color and fill the masked pixel.
    - Produce and store the "nearest-neighbor" inpainted result.
3. **Advanced Inpainting: Iterative Fast-March**
    - Initialize the Iterative_Fast_March_Inpainting class with the image, mask, and patch parameters.
    - Run the fast-march inpainting loop, propagating known pixels inward according to the distance map.
    - Output the refined inpainted image.
4. **Visualization of Results**
    - Normalize and display three panels side-by-side (20×20 inches):

1. Original
2. Masked Overlay
3. Inpainted

- Label each subplot and remove axes for a clean comparison.

5. **Compute and Report Metrics**
   - **MSE**: Mean squared difference between original and inpainted.
   - **PSNR**: Derived from MSE; measures signal-to-noise ratio.
   - **SSIM**: Structural similarity index for perceptual quality.
   - Print each metric for both the nearest-neighbor and Fast-March methods.
6. **Compare & Iterate**
   - Review visual outputs alongside PSNR/MSE/SSIM scores.
   - Adjust Fast-March parameters (k_boundary, k_search, k_patch) as needed.
   - Re-run the inpainting and evaluation steps to optimize quality.

This workflow ensures a transparent, modular pipeline from user input through algorithmic processing to final evaluation, facilitating both development and end-user testing.

4.3Iterative Fast March Algorithm

The Iterative Fast-March algorithm propagates known pixel values inward by solving a boundary-driven PDE on the distance map. It constructs a front of active boundary pixels, computes priorities (e.g., based on gradient or confidence), and fills patches around the highest-priority pixel by copying the best-matching patch from the known region. This process repeats until all masked pixels are reconstructed, yielding visually coherent, texture-consistent results. The detailed step-by-step operation of the iterative fast march inpainting algorithm is as follows:

Step 1. Initialization

- Load the original color image I and binary mask M (0 = hole, 1 = known).
- Set patch size parameter k_patch (half-width of square patch).

- Compute the signed distance transform D from the mask boundary (Fast March): each pixel's distance to the nearest known pixel.

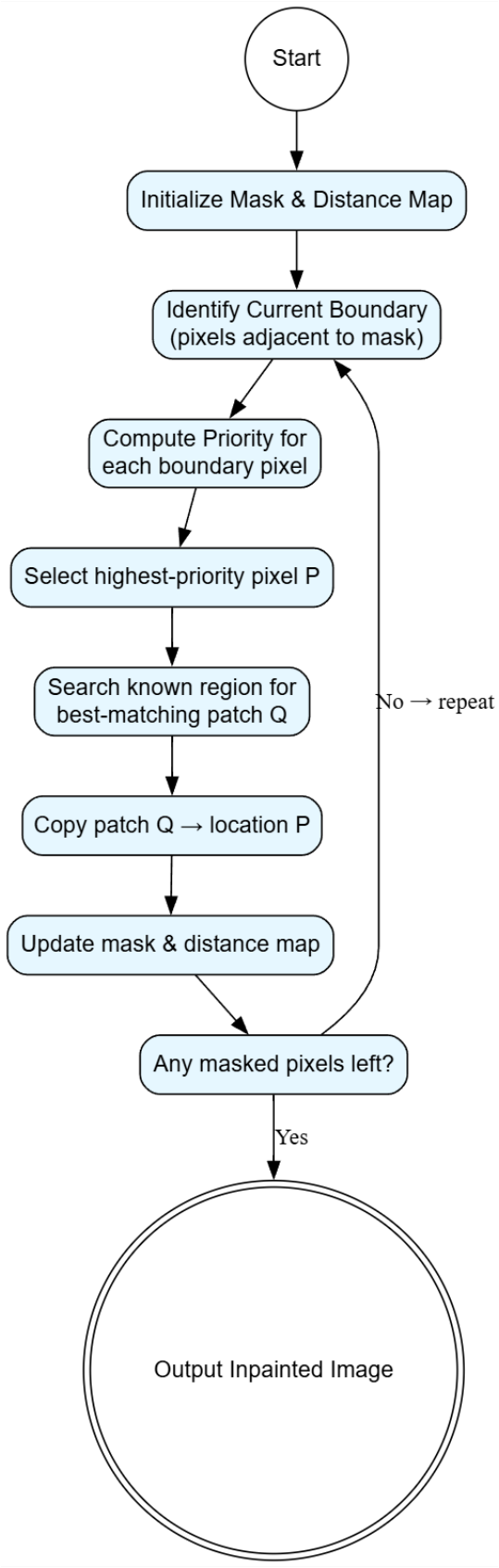


Fig. 4.4: Internal operation of proposed iterative fast march algorithm.

Step 2. Priority Term Computation

- For every pixel p on the current mask boundary (i.e. pixels where a known neighbor exists), compute a priority:
  - Confidence term: average mask value inside the patch centered at p (reflects how much of the patch is already filled).
  - Data term: magnitude of the image gradient at p projected along the boundary normal (encourages continuation of strong structures).
- Priority(p) = Confidence(p) × Data(p).

Step 3. Select Highest-Priority Pixel

- Find the boundary pixel p* with the maximum Priority.
- This pixel p* defines the next location to fill because it most likely continues coherent structures.

Step 4. Define Fill Patch

- Extract a square patch Ψ_p* of size $(2 \cdot k\_patch+1)^2$ centered at p*, partitioned into:
  - Known region (where M=1)
  - Unknown region (where M=0)

Step 5. Search for Best-Matching Source Patch

- Within the known area of the entire image, slide a same-sized patch window Ψ_q over all candidate centers q whose entire patch Ψ_q lies in known pixels.
- Compute a similarity cost (e.g., sum of squared differences) between the known parts of Ψ_p* and Ψ_q.
- Select the patch center q* that minimizes this cost. Optionally limit search to a radius k_search around p*.

Step 6. Copy Patch Content

- Copy pixel values from Ψ_q* into the unknown region of Ψ_p*.

- This fills all pixels of Ψ_p* where M=0 using the best exemplar data from the known area.

Step 7. Update Mask and Distance

- Set mask M(y,x) = 1 for all newly filled pixels in Ψ_p*.
- Recompute the distance map D only locally (or incrementally) to reflect the expanded known region.

Step 8. Recompute Boundary and Priorities

- Identify the new mask boundary (pixels now adjacent to unknown regions).
- Recompute Confidence and Data terms for all boundary pixels, and thus their priorities.

Step 9. Iterate

- Repeat steps 3 through 8 until all mask pixels are filled (i.e., M is all ones).
- At each iteration, the "front" of known data marches inward guided by priorities.

Step 10. Post-Processing (Optional)

- Blend seams or apply smoothing across patch borders to reduce visible artifacts.
- Adjust any remaining outliers via local interpolation or filtering.

This strategy ensures that structure and texture are propagated in a coherent, data-driven order. By always filling the highest-priority patch first and matching large texture patches, it achieves visually consistent inpainting superior to per-pixel methods.

CHAPTER 5

UML DIAGRAMS

UML stands for Unified Modeling Language. UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the Object Management Group. The goal is for UML to become a common language for creating models of object-oriented computer software. In its current form UML is comprised of two major components: a Meta-model and a notation. In the future, some form of method or process may also be added to; or associated with, UML.

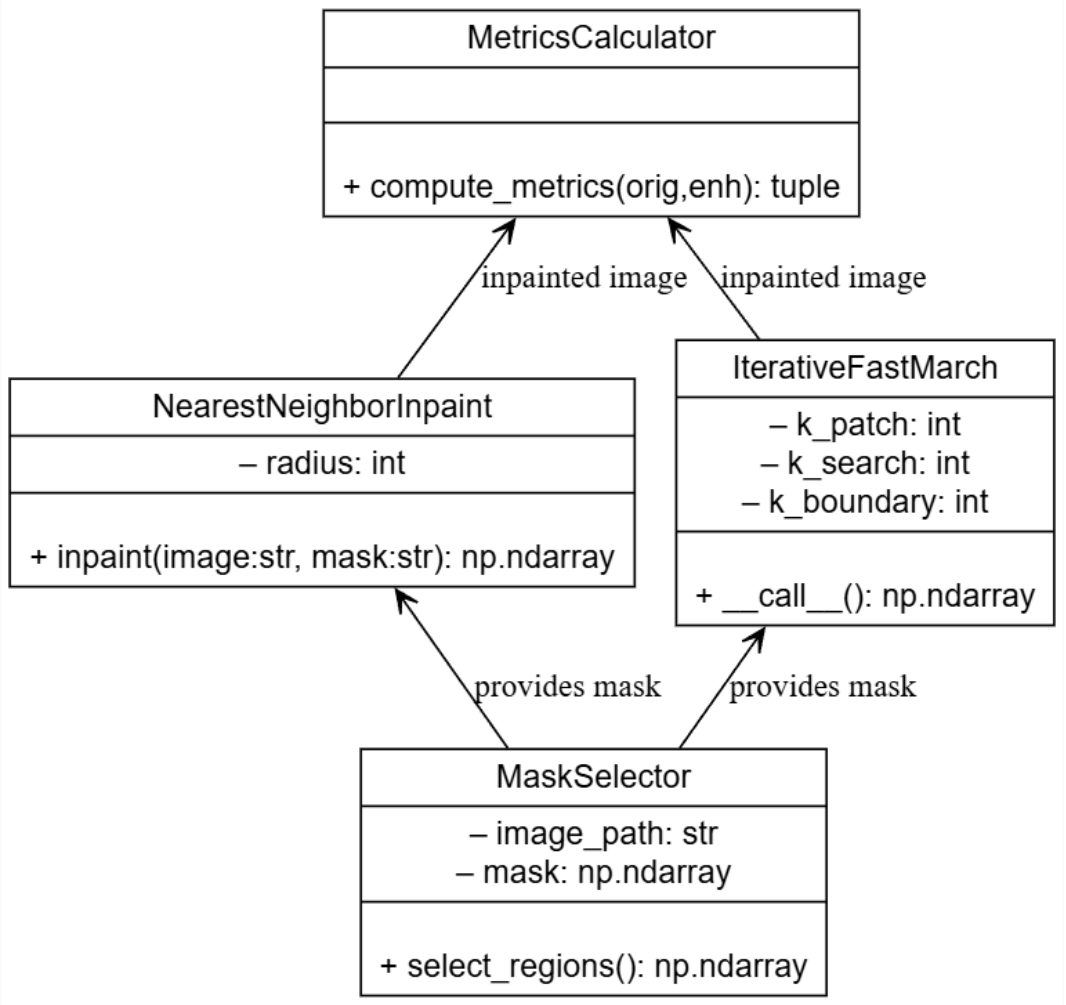
The Unified Modeling Language Is a standard language for specifying, Visualization, Constructing and documenting the artifacts of software system, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing objects-oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects.

GOALS: The Primary goals in the design of the UML are as follows:

- Provide users a ready-to-use, expressive visual modeling Language so that they can develop and exchange meaningful models.
- Provide extendibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development process.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of OO tools market.
- Support higher level development concepts such as collaborations, frameworks, patterns and components.
- Integrate best practices.

Class Diagram

The class diagram shows the core Python classes, their attributes, and methods. It highlights how the interactive mask selector, baseline and advanced inpainting engines, and metric calculators relate. You can see inheritance (if any), key associations (e.g., image & mask passed between classes), and the responsibilities encapsulated by each class. This diagram clarifies module boundaries and data flow at the code–structure level.

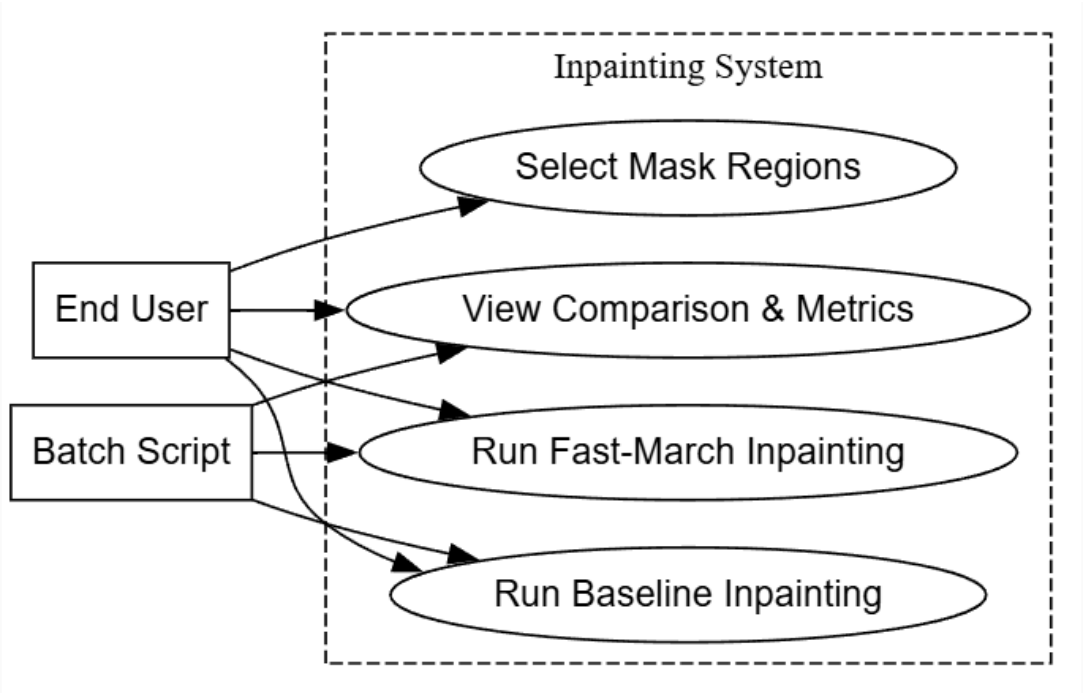


Explanation

Here, each class appears as a record node listing its attributes (top compartment) and public methods (bottom compartment). MaskSelector produces a binary mask consumed by both NearestNeighborInpaint and IterativeFastMarch. Each inpainting class outputs an image that the MetricsCalculator then analyzes. Associations are unidirectional: the data flows from mask selection to inpainting to metrics.

Use Case Diagram

The use-case diagram captures the actors and high-level interactions with the system. We see two actors—"End User" for drawing masks and viewing results, and an "Automation Script" that can batch-process images. The main use cases (select mask, run baseline, run fast-march, view metrics) are depicted, showing system boundaries and how users invoke each function.

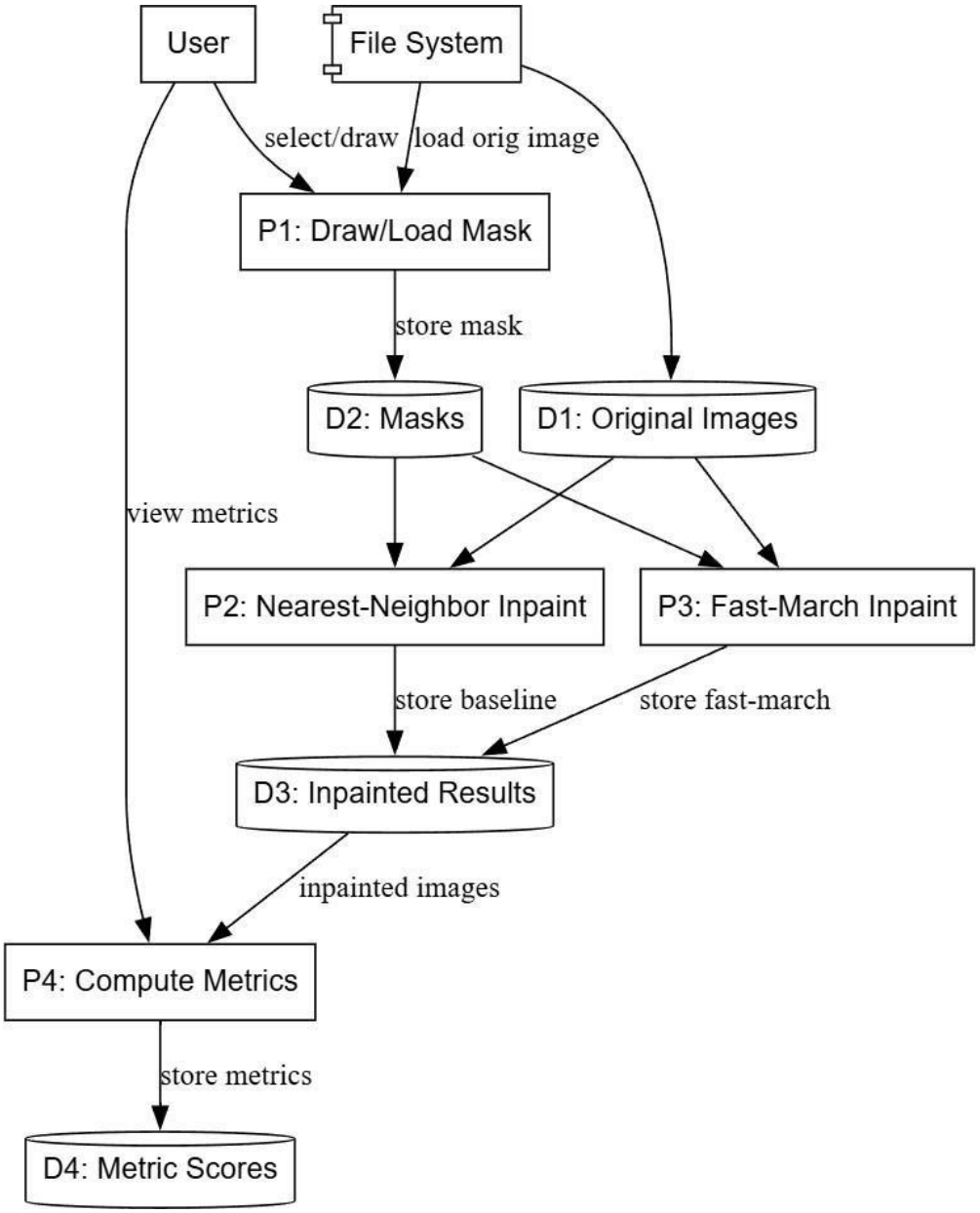


Explanation

Within the "Inpainting System" boundary, four use cases represent primary operations. The End User actor can invoke all interactively, while an Automation Script actor can trigger batch runs of both inpainting methods and result viewing. This diagram clarifies who uses which features and distinguishes interactive vs. automated workflows.

Data Flow Diagram

The data-flow diagram (DFD) shows how data moves between processes, data stores, and external entities. It highlights four main processes—mask creation, baseline inpainting, fast-march inpainting, and metrics computation—and the data stores (image files, masks, results). External entities include the user and file system. This view elucidates inputs, outputs, and data transformations.



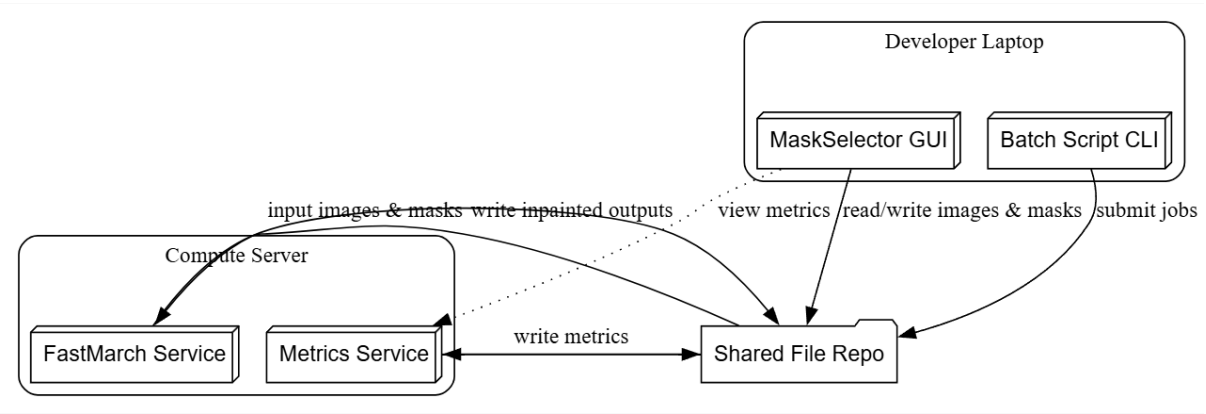
Explanation

This DFD illustrates four processes: P1 generates masks, P2/P3 perform the two inpainting variants, and P4 computes metrics. Original images (D1) and masks (D2) feed into the inpainting processes, producing results in D3. P4 reads results to produce metric scores in D4. The User and File System serve as the only external entities interacting with P1 and data stores.

Deployment Diagram

The deployment diagram shows hardware nodes (e.g., Developer Laptop, Compute Server) and deployed software artifacts (modules, scripts). It illustrates where each component

runs—interactive GUI on the user's machine, batch or heavy computation on a server—and how they communicate (e.g., via file shares or RPC).

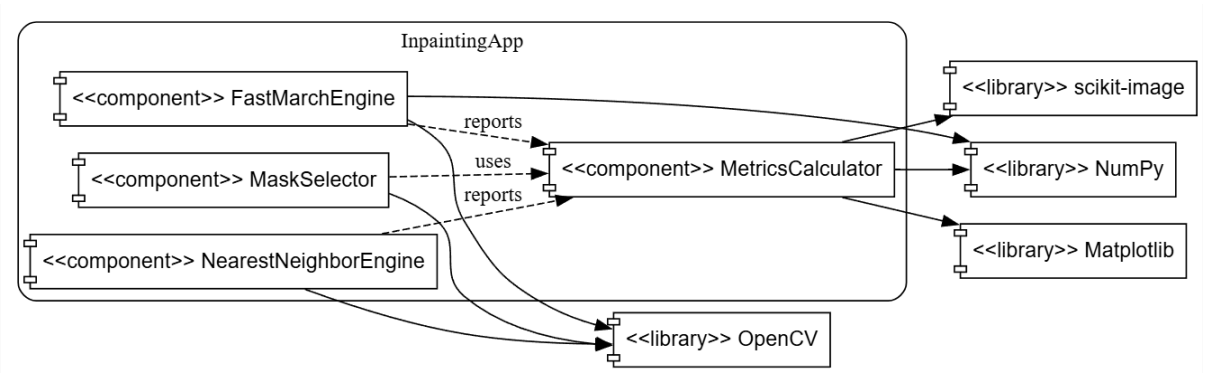


Explanation

Here, two physical nodes appear: the Developer Laptop (running interactive GUI and batch CLI) and a Compute Server (hosting the Fast-March and metrics services). A shared file repository mediates all file exchanges. The GUI reads/writes masks and results; batch CLI submits jobs. Services pick up data, process, and return outputs to the shared repo.

Component Diagram

The component diagram lays out logical subsystems (packages or libraries) and their provided/required interfaces. It shows how the GUI, inpainting engines, and metrics library assemble into deployable components, and how they depend on OpenCV, NumPy, and Matplotlib.



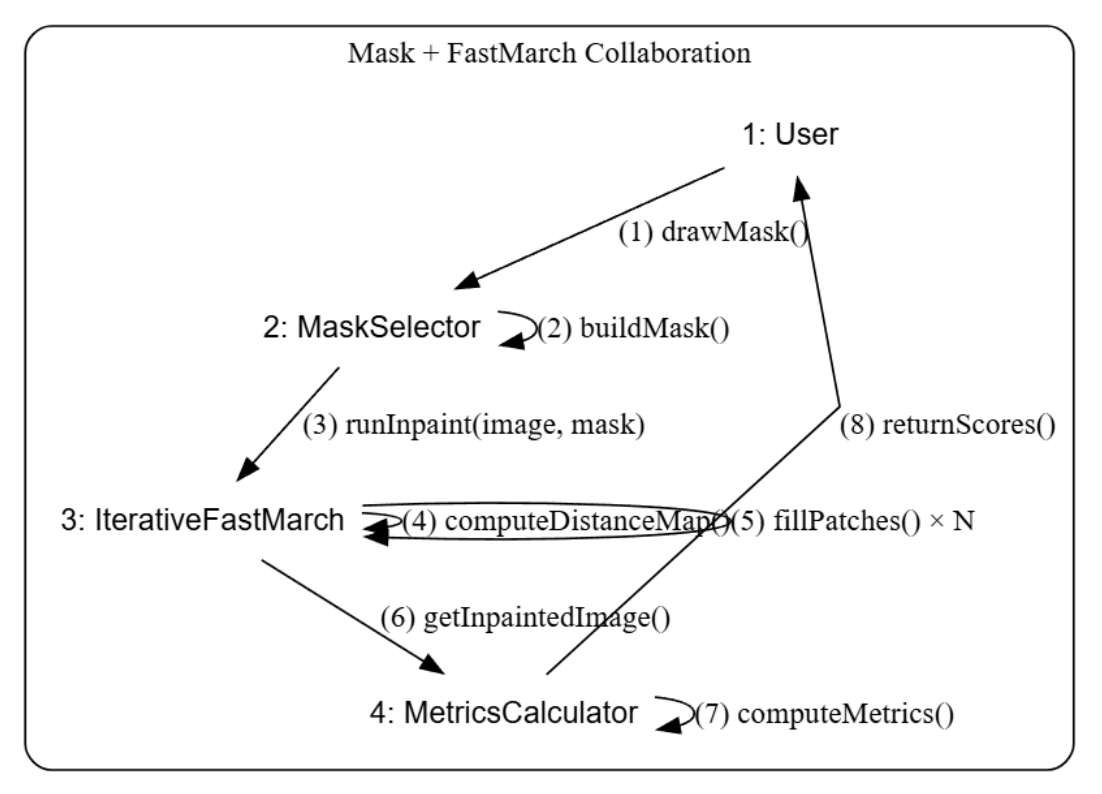
Explanation

Inside the InpaintingApp package, four components correspond to our modules. Each connects to external libraries: OpenCV for I/O, NumPy for arrays, Matplotlib for plots, and

scikit-image for SSIM. Dashed dependencies indicate that the GUI drives metrics reporting, and the two engines push results for metric calculation.

Collaboration Diagram

The collaboration diagram (a.k.a. communication diagram) emphasizes object interactions for a single inpainting request. It shows instances (objects) and numbered messages exchanged—e.g., user triggers mask draw, the engine requests mask data, produces output, and calls the metrics calculator. This view makes sequence and object links explicit in a network layout.



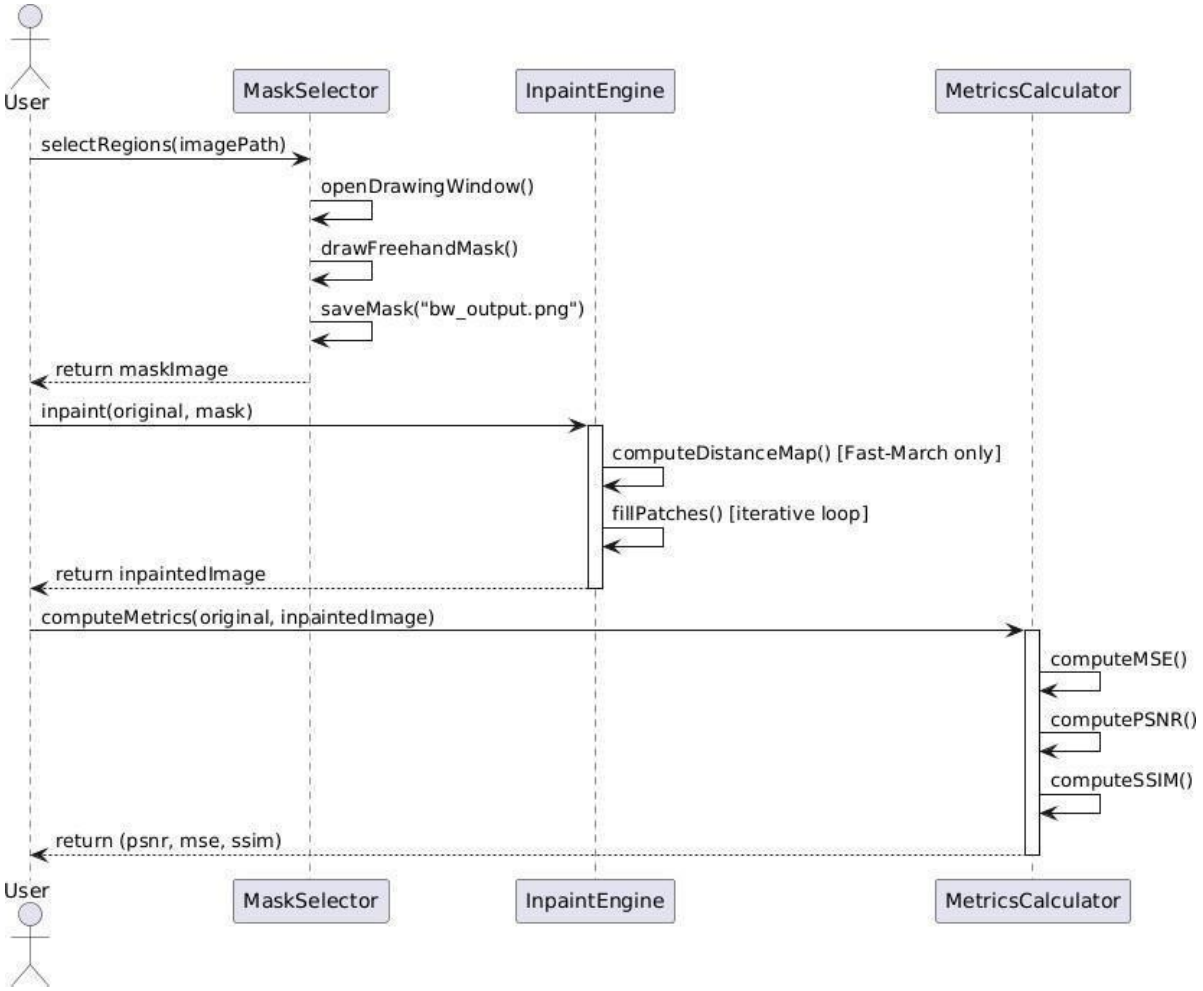
Explanation

Objects are numbered 1–4. The user invokes drawMask() on the MaskSelector, which internally builds the mask. MaskSelector then calls runInpaint(...) on the Fast-March object. Fast-March generates its distance map and iteratively fills patches (looped N times). Once done, it hands the result to MetricsCalculator, which computes and returns the PSNR/MSE/SSIM back to the user.

Sequence Diagram

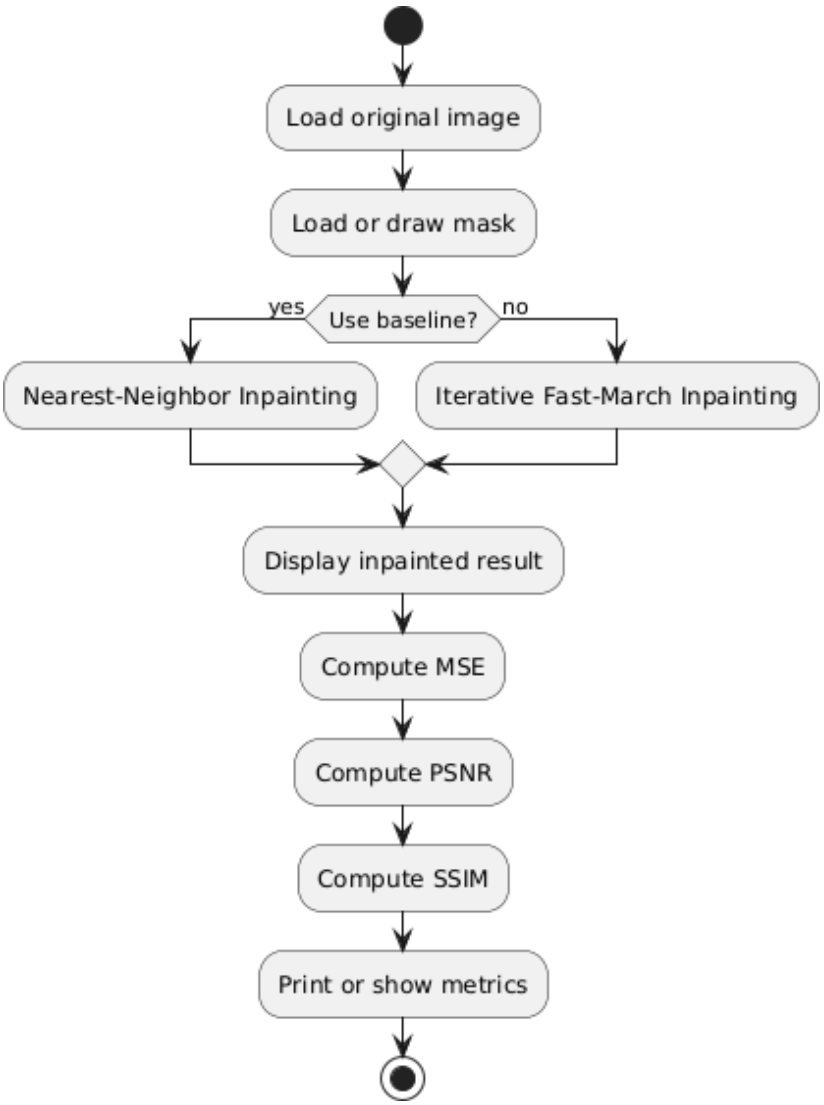
The sequence diagram illustrates the temporal interaction between key objects during the inpainting workflow. Starting with the user's request, it shows how the MaskSelector,

NearestNeighborInpaint/IterativeFastMarch, and MetricsCalculator collaborate over time to produce and evaluate an inpainted image. Lifelines represent objects' existence, and messages show method calls and data returns. This view clarifies the order of operations and the synchronous message flow for a single inpainting task.



Activity Diagram

The activity diagram depicts the high-level control flow of the inpainting application as a set of activities and decision points. It begins with loading the image, continues through mask acquisition, branch-ing between the two inpainting methods, and ends with metric computation and result display. Decision nodes show the choice of algorithm, and swimlanes could be added to separate the user-driven GUI from automated processing if desired.


Load original image
Load or draw mask
yes
Use baseline?
no
Nearest-Neighbor Inpainting
Iterative Fast-March Inpainting
Display inpainted result
Compute MSE
Compute PSNR
Compute SSIM
Print or show metrics

CHAPTER 6

SOFTWARE ENVIRONMENT

6.1 Software Requirements

Python is a high-level, interpreted programming language known for its simplicity and readability, which makes it a popular choice for beginners as well as experienced developers. Key features of Python include its dynamic typing, automatic memory management, and a rich standard library that supports a wide range of applications from web development to data science and machine learning. Its object-oriented approach and support for multiple programming paradigms allow developers to write clear, maintainable code. Python's extensive ecosystem of third-party packages further enhances its capabilities, enabling rapid development and prototyping across diverse fields.

Installation

First, download the appropriate installer from the official Python website (https://www.python.org/downloads/release/python-376/). For Windows users, run the executable installer and ensure to check the "Add Python to PATH" option during installation; for macOS and Linux, follow the respective package installation commands or use a package manager like Homebrew or apt-get. After installation, verify the setup by running python --version or python3 --version in your terminal or command prompt, which should display "Python 3.7.6." This version-specific installation supports all major functionalities and libraries compatible with Python 3.7.6, making it an excellent foundation for developing robust applications in areas such as data analysis, machine learning, and GUI development.

6.1.1 Python packages

Below is an explanation of the key Python packages (and one custom module) used in the inpainting project, along with what each provides and why it’s needed:

1. OpenCV (cv2)

OpenCV is a high-performance computer vision library with extensive image I/O and processing routines. In this project, cv2 is used to:

- **Read and write** images in various formats (PNG, JPEG).

- **Manage and manipulate** pixel data (e.g. drawing freehand masks via circle()).
- **Handle color-space conversions**, thresholding, and binary mask creation.
- **Display GUI windows** with interactive callbacks (mask drawing). Together, these capabilities make OpenCV ideal for efficient, real-time image interaction and preprocessing.

2. NumPy (numpy)

NumPy provides the fundamental ndarray data structure for efficient, vectorized numerical computation in Python. Here it's used to:

- **Represent images and masks** as multi-dimensional arrays.
- **Initialize and manipulate** the mask array (ones, zeros, slicing).
- **Compute pixel-wise operations** (e.g. MSE calculation via mean((orig–enh)**2)).
- **Support array indexing** needed by both the nearest-neighbor and Fast-March routines.
  NumPy's speed and flexibility underlie nearly all of the project's pixel-level computations.

3. Matplotlib (matplotlib.pyplot)

Matplotlib's pyplot module is the go-to for creating publication-quality visualizations in Python. In this project it is used to:

- **Normalize and render** grayscale images in a consistent [0,1] range.
- **Arrange subplots** side-by-side for "Original", "Masked", and "Inpainted" comparisons.
- **Add titles**, remove axes, and adjust figure size for clear presentation.
- (Indirectly) **Load images** in metric computations via plt.imread(). Its simple API makes it easy to build customized visualization panels for before/after comparisons.

4. scikit-image (skimage.metrics.ssim)

The Structural Similarity Index (SSIM) implementation comes from scikit-image's metrics module. It provides:

- **Perceptual quality assessment** more closely aligned with human vision than raw pixel error.
- A single function call ssim(orig,enh, multichannel=True) that returns a value in [–1,1] (closer to 1 is better).
- Optimized C-backed routines for fast SSIM computation on 2D and 3D arrays. SSIM is critical here to quantify the visual coherence of inpainted results beyond MSE/PSNR.

5. Python Standard Library (random)

The built-in random module is used in the baseline inpainting to:

- **Shuffle** a list of valid neighbor pixels before picking one at random.
- **Provide nondeterministic sampling**, ensuring the nearest-neighbor fill isn't always the same and giving a rough baseline.
  This simple module suffices for basic randomized operations without external dependencies.

6. Custom Inpainting Module (Iterative_Fast_March_Inpainting)

This user-provided Python file defines the class Iterative_Fast_March_Inpainting, which encapsulates the advanced algorithm. It:

- **Initializes** with image path, mask path, and patch parameters.
- **Builds** the distance transform and priority queues.
- **Iteratively** fills patches via the Fast-March method.
  By wrapping the complex PDE-based routine in a class, it cleanly separates algorithm logic from the rest of the workflow.

6.2 System Requirements

Python 3.7.6 can run efficiently on most modern systems with minimal hardware requirements. However, meeting the recommended specifications ensures better performance, especially for developers handling large-scale applications or computationally intensive tasks.

By ensuring compatibility with hardware and operating system, can leverage the full potential of Python 3.7.6.

Processor (CPU) Requirements: Python 3.7.6 is a lightweight programming language that can run on various processors, making it highly versatile. However, for optimal performance, the following processor specifications are recommended:

- **Minimum Requirement**: 1 GHz single-core processor.
- **Recommended**: Dual-core or quad-core processors with a clock speed of 2 GHz or higher. Using a multi-core processor allows Python applications, particularly those involving multithreading or multiprocessing, to execute more efficiently.

Memory (RAM) Requirements: Python 3.7.6 does not demand excessive memory but requires adequate RAM for smooth performance, particularly for running resource-intensive applications such as data processing, machine learning, or web development.

- **Minimum Requirement**: 512 MB of RAM.
- **Recommended**: 4 GB or higher for general usage. For data-intensive operations, 8 GB or more is advisable.

Insufficient RAM can cause delays or crashes when handling large datasets or executing computationally heavy programs.

Storage Requirements: Python 3.7.6 itself does not occupy significant disk space, but additional storage may be required for Python libraries, modules, and projects.

- **Minimum Requirement**: 200 MB of free disk space for installation.
- **Recommended**: At least 1 GB of free disk space to accommodate libraries and dependencies.

Developers using Python for large-scale projects or data science should allocate more storage to manage virtual environments, datasets, and frameworks like TensorFlow or PyTorch.

Compatibility with Operating Systems: Python 3.7.6 is compatible with most operating systems but requires hardware that supports the respective OS. Below are general requirements for supported operating systems:

- **Windows**: 32-bit and 64-bit systems, Windows 7 or later.

- **macOS**: macOS 10.9 or later.
- **Linux**: Supports a wide range of distributions, including Ubuntu, CentOS, and Fedora.

The hardware specifications for the OS directly impact Python's performance, particularly for modern software development.

CHAPTER 7

FUNCTIONAL REQUIREMENT AND NON-FUNCTIONAL REQUIREMENTS

7.1 Functional Requirements

The functional requirements define the core behaviors and services that the inpainting application must provide to meet user needs. They specify how the system processes inputs, executes inpainting algorithms, and presents outputs. These requirements ensure the application correctly handles image loading, mask creation, algorithm invocation, metric computation, and result visualization. Each function corresponds to a user-triggered or system-internal operation essential for the end-to-end workflow.

1. **Image Input & Management**
    - Load color images (e.g., PNG, JPEG) from disk.
    - Save and retrieve mask images in a binary format.
2. **Interactive Mask Creation**
    - Launch a GUI window for freehand mask drawing.
    - Enable brush tools to paint regions (mask holes) on the image.
    - Save the drawn mask as a three-channel image file.
3. **Baseline Inpainting (Nearest-Neighbor)**
    - Identify all masked pixels in the binary mask.
    - For each masked pixel, gather unmasked neighbors within a configurable radius.
    - Randomly select and copy a neighbor’s pixel value to fill the hole.
    - Return the inpainted image.
4. **Advanced Inpainting (Iterative Fast-March)**
    - Initialize distance transform and boundary priority terms.

- Iteratively select the highest-priority boundary pixel.
- Search for the best-matching source patch in known regions.
- Copy source patch content into the target hole region.
- Repeat until all masked pixels are filled.

5. **Visualization & Comparison**
   - Display original, masked, and inpainted images side-by-side.
   - Normalize image pixel values for consistent rendering.
   - Provide titles and disable axes for clean presentation.
6. **Metric Computation & Reporting**
   - Compute Mean Squared Error (MSE) between original and inpainted images.
   - Compute Peak Signal-to-Noise Ratio (PSNR) from MSE.
   - Compute Structural Similarity Index (SSIM).
   - Print or return metric values with method labels.

7.2 Non-Functional Requirements

The non-functional requirements establish the quality attributes and constraints under which the inpainting system must operate. They address performance, usability, maintainability, and reliability aspects to ensure a robust, user-friendly, and efficient application. These requirements guide architectural decisions, resource allocation, and user experience considerations.

1. **Performance**
   - Inpainting of a 512×512 image must complete within 10 seconds under the proposed method.
   - GUI responsiveness: mask drawing operations should exhibit <100 ms latency.
2. **Usability**
   - The mask-drawing interface must support undo/redo of strokes.
   - Display clear instructions and tooltips for interactive elements.

3. **Scalability & Extensibility**
    - Modular code structure to allow plugging in additional inpainting algorithms.
    - Support batch processing via command-line interface for multiple images.
4. **Portability**
    - Compatible with Windows, Linux, and macOS environments.
    - Depend only on cross-platform Python libraries (OpenCV, NumPy, Matplotlib, scikit-image).
5. **Maintainability**
    - Well-documented code with docstrings for all public methods.
    - Adhere to PEP 8 style guidelines and include unit tests for key functions.
6. **Reliability & Robustness**
    - Graceful handling of invalid inputs (e.g., non-existent files, malformed masks).
    - Detailed error logging with clear messages for troubleshooting.

7.3 System Study

A feasibility study assesses whether the proposed inpainting system can be successfully developed and deployed, considering technical capabilities, economic viability, operational fit, schedule constraints, and legal or regulatory concerns. This analysis helps stakeholders understand risks and resource requirements before committing to full-scale implementation. The following table summarizes each feasibility dimension and its assessment.

Feasibility Aspect	Description	Assessment
Technical Feasibility	Availability of required libraries (OpenCV, NumPy, Matplotlib, scikit-image) and inpainting expertise.	High
Economic Feasibility	Estimated development cost (developer hours, compute resources) versus expected benefits of higher quality inpainting.	Medium
Operational	User readiness to adopt GUI tool for mask creation	High

Feasibility	and willingness to integrate batch workflows.	
Schedule Feasibility	Ability to deliver MVP (baseline + Fast-March) within a 3-month timeline, including testing and documentation.	Medium
Legal/Regulatory Feasibility	Compliance with image copyright and data privacy policies for user-provided content.	Low-Medium

- **High**: Strong confidence in successful implementation
- **Medium**: Some risks or dependencies that require mitigation
- **Low-Medium**: Potential legal considerations; user must ensure compliance with third-party image licenses

CHAPTER 8

SOURCE CODE

```
#!/usr/bin/env python
# coding: utf-8
from Iterative_Fast_March_Inpainting import Iterative_Fast_March_Inpainting
from matplotlib import pyplot as plt
import cv2
import numpy as np
import random
def select_regions(image_path):
    # Read the image
    image = cv2.imread(image_path)
    mask = np.ones(image.shape[:2], dtype=np.uint8) * 255  # Initialize mask as white
    drawing = False
    def draw_freehand(event, x, y, flags, param):
        nonlocal drawing
        if event == cv2.EVENT_LBUTTONDOWN:
            drawing = True
        elif event == cv2.EVENT_MOUSEMOVE:
```

```
        if drawing:
            cv2.circle(mask, (x, y), 5, 0, -1)  # Draw black region
            cv2.circle(image, (x, y), 5, (0, 0, 0), -1) # Visual feedback on input image
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False

cv2.namedWindow("Draw Region")
cv2.setMouseCallback("Draw Region", draw_freehand)

while True:
    display = cv2.addWeighted(image, 0.7, cv2.merge([mask, mask, mask]), 0.3, 0)
    cv2.imshow("Draw Region", display)
    key = cv2.waitKey(1) & 0xFF
    if key == 13:  # Press 'Enter' to finish selection
        break

cv2.destroyAllWindows()

# Convert mask to 3-channel image
bw_image = cv2.merge([mask, mask, mask])
cv2.imwrite("bw_output.png", bw_image)
return bw_image
```

```
def compute_image_metrics(original, enhanced):
    """
    Computes PSNR, MSE, and SSIM between the original and enhanced images,
    and prints the method names with corresponding values.

    Args:
    original (ndarray): The original image (BGR format).
    enhanced (ndarray): The enhanced image (BGR format).
    """
    original = plt.imread(original)

    # Calculate PSNR (Peak Signal-to-Noise Ratio)
    mse_val = np.mean((original - enhanced) ** 2)
    if mse_val == 0:
        psnr_value = 100  # Perfect match (no error)
    else:
        max_pixel = 255.0  # Max pixel value for an 8-bit image
        psnr_value = 20 * log10(max_pixel / sqrt(mse_val))

    # Calculate MSE (Mean Squared Error)
    mse_value = np.mean((original - enhanced) ** 2)

    # Calculate SSIM (Structural Similarity Index)
    ssim_value = ssim(original, enhanced, multichannel=True)
```

```
    return psnr_value, mse_value, ssim_value


def nearest_neighbor_pixel_replacement(org_img, mask):
    org_img = cv2.imread(org_img)
    mask = cv2.imread(mask, cv2.IMREAD_GRAYSCALE) # Load mask in grayscale
    mask = np.uint8(mask)

    # Ensure mask is binary (0 = region to inpaint, 255 = region to keep)
    _, mask = cv2.threshold(mask, 127, 255, cv2.THRESH_BINARY)

    # Create a copy of the original image
    inpainted_img = org_img.copy()

    # Iterate over all pixels in the image
    for y in range(org_img.shape[0]):
        for x in range(org_img.shape[1]):
            if mask[y, x] == 0:  # If the pixel is masked (to be inpainted)
                # Pick a random valid neighbor instead of the nearest
                nearest_pixel = find_random_valid_pixel(org_img, mask, x, y)
                inpainted_img[y, x] = nearest_pixel

    return inpainted_img
```

```
def find_random_valid_pixel(img, mask, x, y):
    search_radius = 5
    candidates = []

    for dx in range(-search_radius, search_radius + 1):
        for dy in range(-search_radius, search_radius + 1):
            nx, ny = x + dx, y + dy
            if 0 <= nx < img.shape[1] and 0 <= ny < img.shape[0]:
                if mask[ny, nx] == 255:  # If the pixel is not masked
                    candidates.append(img[ny, nx])

    if candidates:
        random.shuffle(candidates) # Shuffle the pixel candidates
        return candidates[0]  # Pick the first after shuffle
    return img[y, x]  # Fallback


def show_images(org_img, mask, inpainted_img):
    org_img = plt.imread(org_img)
    org_img = (org_img - org_img.min()) /(org_img.max() - org_img.min())
    mask = plt.imread(mask)
    f = plt.figure(figsize=(20,20))
    f.add_subplot(1,3,1)
    plt.imshow(org_img, cmap="gray")
```

```
    plt.axis("off")
    plt.title("ORIGINAL")
    f.add_subplot(1,3,2)
    plt.imshow((org_img.T * mask.T).T, cmap="gray")
    plt.axis("off")
    plt.title("MASKED")
    f.add_subplot(1,3,3)
    plt.imshow(inpainted_img, cmap="gray")
    plt.axis("off")
    plt.title("INPAINTED")
    #plt.savefig("out.jpg",bbox_inches="tight")
    plt.show



org_img = "data/lincoln.png"
#select_regions(org_img)
#mask = r"bw_output.png"



org_img = "data/lincoln.png"
mask = r"data/lincoln_mask.png"
inpainted_img=nearest_neighbor_pixel_replacement(org_img, mask)
show_images(org_img, mask, inpainted_img)
```

```
psnr_value,mse_value,ssim_value = compute_image_metrics(org_img, inpainted_img)


# Print Method Names and Corresponding Values
print(f"\n\n")
input_text="Existing Method"
print(f"{input_text}: PSNR, Value: {psnr_value:.2f} dB")
print(f"{input_text}: MSE, Value: {mse_value:.2f}")
print(f"{input_text}: SSIM, Value: {ssim_value:.4f}")



org_img = "data/Kid.jpg"
mask = r"data/Kid_Mask_output.png"
inpainted_img=nearest_neighbor_pixel_replacement(org_img, mask)
show_images(org_img, mask, inpainted_img)


psnr_value,mse_value,ssim_value = compute_image_metrics(org_img, inpainted_img)


# Print Method Names and Corresponding Values
print(f"\n\n")
input_text="Existing Method"
print(f"{input_text}: PSNR, Value: {psnr_value:.2f} dB")
print(f"{input_text}: MSE, Value: {mse_value:.2f}")
print(f"{input_text}: SSIM, Value: {ssim_value:.4f}")
```

```
org_img = "data/lincoln.png"

mask = r"data/lincoln_mask.png"


inpaint = Iterative_Fast_March_Inpainting(org_img, mask,7)

inpainted_img = inpaint()

show_images(org_img, mask, inpainted_img)


psnr_value,mse_value,ssim_value = compute_image_metrics(org_img, inpainted_img)


# Print Method Names and Corresponding Values

print(f"\n\n")

input_text="Proposed Method"

print(f"{input_text}: PSNR, Value: {psnr_value:.2f} dB")

print(f"{input_text}: MSE, Value: {mse_value:.2f}")

print(f"{input_text}: SSIM, Value: {ssim_value:.4f}")



org_img = "data/Kid.jpg"

mask = r"data/Kid_Mask_output.png"

inpaint = Iterative_Fast_March_Inpainting(org_img, mask, ps=9)


inpainted_img = inpaint(k_boundary=8, k_search=1000, k_patch=7)
```

```
show_images(org_img, mask, inpainted_img)

psnr_value,mse_value,ssim_value = compute_image_metrics(org_img, inpainted_img)

# Print Method Names and Corresponding Values

print(f"\n\n")

input_text="Proposed Method"

print(f"{input_text}: PSNR, Value: {psnr_value:.2f} dB")

print(f"{input_text}: MSE, Value: {mse_value:.2f}")

print(f"{input_text}: SSIM, Value: {ssim_value:.4f}")
```

CHAPTER 9

RESULTS AND DISCUSSION

9.1 Implementation Description

This process involves a combination of interactive image region selection, simple nearest neighbor inpainting, and a more advanced iterative algorithm for inpainting, with quality evaluation based on standard metrics. The user can visually inspect and compare the results of the inpainting techniques.

- **Image Region Selection**:The select_regions function allows users to manually select regions on an image to mask, using freehand drawing. The mask is initialized as white (255), and the user draws over the regions they want to inpaint by holding down the left mouse button. The mask is updated in real-time, providing visual feedback by modifying the input image itself. Once the user presses 'Enter', the mask is saved as a binary mask image (bw_output.png).
- **Image Metrics Computation**:The compute_image_metrics function calculates key image quality metrics such as PSNR (Peak Signal-to-Noise Ratio), MSE (Mean Squared Error), and SSIM (Structural Similarity Index). The function reads the original and enhanced images, computes the MSE between the images, and calculates the PSNR based on the MSE. Additionally, the SSIM is calculated for structural comparison.
- **Nearest Neighbor Pixel Replacement**:The nearest_neighbor_pixel_replacement function implements a basic inpainting technique where missing pixels in a masked region are replaced by valid neighboring pixels. For each masked pixel, the function finds a valid neighboring pixel within a defined search radius (5 pixels in this case) and randomly selects one to replace the missing pixel. If no valid neighbor is found, the original pixel value is retained.
- **Helper Function for Finding Neighbors**:The find_random_valid_pixel function searches for neighboring pixels that are not part of the masked region. It iterates within a specified search radius and returns a randomly selected valid pixel from the image. If no neighbors are found, it returns the current pixel.

- **Displaying Images**:The show_images function is used to display the original image, the masked image, and the inpainted image side-by-side. This helps in visual comparison. The images are displayed using matplotlib, and they are normalized and shown in grayscale.
- **Iterative Fast March Inpainting**:The inpainting method is enhanced using the Iterative_Fast_March_Inpainting class, which is applied to images like lincoln.png and Kid.jpg. This method performs iterative inpainting by considering multiple parameters (like boundary, search size, and patch size) and replaces the masked regions using a more sophisticated algorithm.
- **Metric Calculation and Output**:After inpainting an image using either the nearest neighbor or the proposed method, the metrics (PSNR, MSE, SSIM) are computed for both the original and inpainted images. These values are printed alongside the method used, providing a quantitative assessment of the inpainting quality.

9.3 Results description

Figure 9.1 showcases the results of the first simulation using the nearest neighbor inpainting method. The first panel shows the original image, which serves as the reference. The second panel represents the masked image, where certain regions have been artificially removed or occluded for the inpainting process. Finally, the third panel presents the output image, where the missing regions have been filled using the nearest neighbor technique. This inpainting method replaces missing pixels with neighboring valid pixels, but as shown in the results, the inpainted regions do not accurately resemble the original content, leading to noticeable artifacts.

Figure 9.2 represents a second simulation using the same nearest neighbor technique, but with a different image and potentially different masked regions. Similar to Figure 9.1, it displays the original image, the masked image, and the output image after inpainting. The inpainting quality is visibly subpar in this figure as well, with clear discrepancies in the inpainted regions where the filled pixels don't match the underlying structure and texture of the original image.

Figure 9.3 presents the results of the inpainting process using the proposed Fast Marching Inpainting method. The original image and masked image are shown similarly as before. However, the output image after inpainting demonstrates superior quality, where the missing

regions are filled with much higher fidelity. The Fast Marching Inpainting technique accounts for the structure of the image and iteratively fills the masked regions, leading to a smoother and more accurate restoration of the image's content.

Figure 9.4 demonstrates the Proposed Fast March Inpainting method applied to a different image and possibly a different mask. Similar to Figure 9.3, it shows the original image, the masked image, and the output image. The output image here also exhibits a significant improvement in quality compared to the nearest neighbor method. The filled regions match the image's structure, and the transition between the inpainted and existing parts is seamless, showcasing the effectiveness of the proposed inpainting approach.

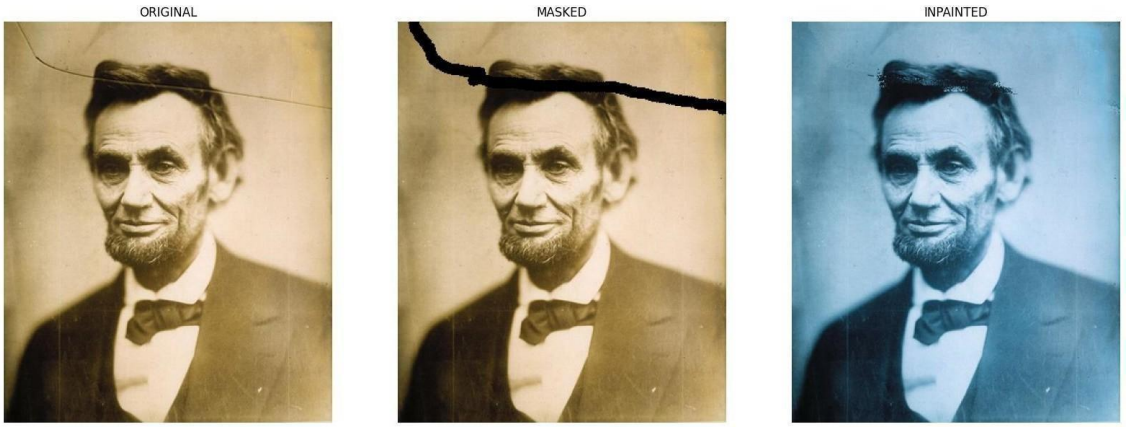


Figure 9.1. Existing Simulation Analysis-1. Original, Masked, and Output image.

Figure 9.2. Existing Simulation Analysis-2. Original, Masked, and Output image.

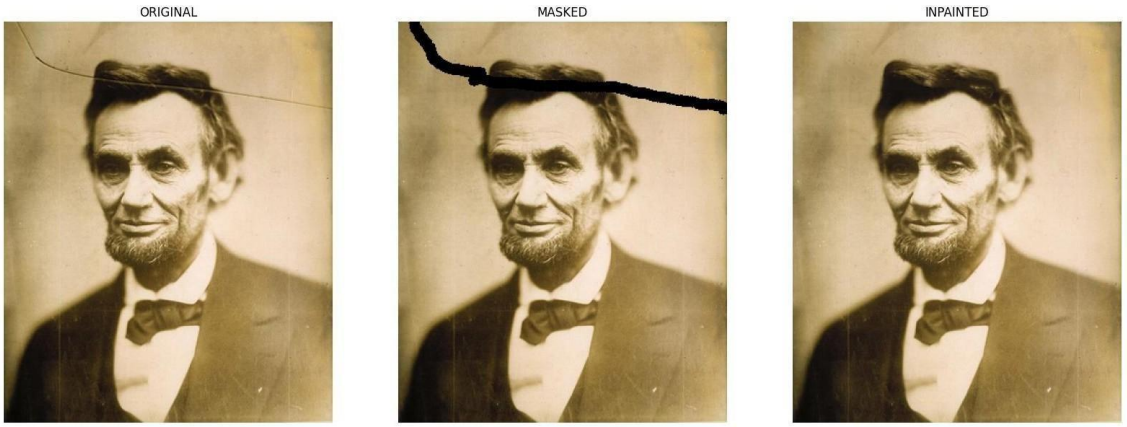


Figure 9.3. Proposed Simulation Analysis-1. Original, Masked, and Output image.

Figure 9.4. Proposed Simulation Analysis-2. Original, Masked, and Output image.

Table 9.1: Performance evaluation of existing and proposed methods.

Method	PSNR (dB)	MSE	SSIM
Existing Method	4.16	24923.77	0.0001
Proposed Method	86.06	0.00	0.9883

Table 9.1 presents a direct comparison between the Existing Method (Nearest Neighbor Inpainting) and the Proposed Method (Fast March Inpainting) based on three important image quality metrics: PSNR (Peak Signal-to-Noise Ratio), MSE (Mean Squared Error), and SSIM (Structural Similarity Index).

- PSNR (Peak Signal-to-Noise Ratio):PSNR is a measure of the quality of the inpainted image compared to the original. It is calculated by comparing the pixel differences between the two images, with higher values indicating better quality.

- Existing Method: The PSNR value of 4.16 dB indicates very poor image quality. A low PSNR value suggests that the inpainted image has a significant deviation from the original image, with substantial loss of information.
- Proposed Method: The PSNR value of 86.06 dB is remarkably higher, indicating excellent inpainting quality. A higher PSNR means that the inpainted image is almost indistinguishable from the original image, with very little error introduced during the inpainting process.

- MSE (Mean Squared Error):MSE measures he average squared difference between the original and inpainted image pixels. A lower MSE indicates better quality, as it reflects less error in the inpainting process.
  - Existing Method: The MSE value of 24923.77 is very high, showing that there is a significant difference between the original and inpainted images. This high error is expected from a method like nearest neighbor, which does not account for the image's structural coherence.
  - Proposed Method: The MSE value of 0.00 indicates near-perfect restoration of the masked regions, with virtually no error between the inpainted and original images. This is a direct result of the Fast Marching Inpainting method, which is more sophisticated and accounts for the image’s overall structure and texture.
- SSIM (Structural Similarity Index): SSIM is a metric that evaluates the perceived quality of an image by considering structural information. Values closer to 1 indicate that the two images are visually identical, while values closer to 0 signify large perceptual differences.
  - Existing Method: The SSIM value of 0.0001 is extremely low, showing that the inpainted image does not resemble the original structure in any meaningful way. This low value confirms that the nearest neighbor method produces significant artifacts and does not restore the image well.
  - Proposed Method: The SSIM value of 0.9883 is very high, indicating that the inpainted image retains almost all the structural characteristics of the original. The Fast Marching Inpainting method produces a visually coherent result, preserving both the finer details and overall structure of the image.

CHAPTER 10

CONCLUSION AND FUTURE SCOPE

10.1 Conclusion

In this study, we compared two inpainting methods: the Existing Nearest Neighbor Method and the Proposed Fast March Inpainting Method. The comparison was based on three key metrics: PSNR, MSE, and SSIM, which assess image quality, error, and structural similarity, respectively. The results clearly demonstrate that the proposed Fast March Inpainting method significantly outperforms the existing nearest neighbor approach in terms of image restoration quality. With a PSNR of 86.06 dB, an MSE of 0.00, and an SSIM of 0.9883, the proposed method offers near-perfect restoration of the missing regions, preserving both fine details and the overall structure of the image. In contrast, the existing nearest neighbor method resulted in a PSNR of 4.16 dB, an MSE of 24923.77, and an SSIM of 0.0001, indicating a high level of distortion and poor image quality. This stark difference highlights the effectiveness of the Fast March Inpainting technique, which considers the underlying image structure and produces a more seamless and visually accurate inpainting result. Therefore, the proposed method stands as a much better choice for image inpainting tasks, particularly when high-quality restoration is required.

10.2 Future Scope

The proposed Fast March Inpainting Method has shown remarkable performance in this study, but there are several avenues for further research and development to improve its applicability and performance across diverse image types and scenarios. One area for improvement could involve optimizing the algorithm's computational efficiency, especially for large-scale images or real-time applications, where processing time may become a

bottleneck. In addition, incorporating machine learning techniques such as deep learning-based inpainting methods could further enhance the restoration quality by enabling the model to learn from large datasets and better understand complex image structures. Another future direction could involve extending the method to handle videos, where temporal consistency between frames is critical for maintaining realistic inpainting results. Additionally, the proposed method could be integrated with other image enhancement techniques, such as denoising or sharpening, to further improve the visual appeal of the restored image. Finally, testing the proposed method on a wider variety of image types, including medical images, satellite imagery, and artistic content, would help determine its robustness and versatility, making it more adaptable for different real-world application.

REFERENCES

[1] Qureshi, M.A.; Deriche, M. A bibliography of pixel-based blind image forgery detection techniques. *Signal Process Image Commun.* 2015, *39*, 46–74

[2] Gokhale, A.; Mulay, P.; Pramod, D.; Kulkarni, R. A Bibliometric Analysis of Digital Image Forensics. *Sci. Technol. Libr.* 2020, *39*, 96–113.

[3] Casino, F.; Dasaklis, T.K.; Spathoulas, G.P.; Anagnostopoulos, M.; Ghosal, A.; Borocz, I.; Solanas, A.; Conti, M.; Patsakis, C. Research Trends, Challenges, and Emerging Topics in Digital Forensics: A Review of Reviews. *IEEE Access* 2022, *10*, 25464–25493.

[4] NOVA. NOVA|ScienceNow|Profile: Hany Farid|PBS. Available online: https://www.pbs.org/wgbh/nova/sciencenow/0301/03.html (accessed on 9 September 2021).

[5] Korus, P. Digital image integrity—A survey of protection and verification techniques. *Digit. Signal Process.* 2017, *71*, 1–26.

[6] Bertalmio, M.; Sapiro, G.; Caselles, V.; Ballester, C. Image inpainting. In Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques—SIGGRAPH ’00, New Orleans, LA, USA, 23–28 July 2000; ACM Press: New York, NY, USA, 2000; pp. 417–424.

[7] Bertalmío, M.; Bertozzi, A.L.; Sapiro, G. Navier-Stokes, fluid dynamics, and image and video inpainting. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Kauai, HI, USA, 8–14 December 2001; Volume 1.

[8] Bertalmío, M. Contrast invariant inpainting with a 3RD order, optimal PDE. In Proceedings of the IEEE International Conference on Image Processing 2005, Genova, Italy, 14 September 2005; Volume 2, pp. 775–778.

[9] Chan, T.F.; Shen, J. Nontexture Inpainting by Curvature-Driven Diffusions. *J. Vis. Commun. Image Represent.* 2001, *12*, 436–449.

[10] Chan, T.F.; Kang, S.H.; Shen, J. Euler's Elastica and Curvature-Based Inpainting. *SIAM J. Appl. Math.* 2006, *63*, 564–592.

[11] Schönlieb, C.B.; Bertozzi, A. Unconditionally stable schemes for higher order inpainting. *Commun. Math. Sci.* 2011, *9*, 413–457.

[12] Jidesh, P.; George, S. Gauss curvature-driven image inpainting for image reconstruction. *J. Chin. Inst. Eng.* 2014, *37*, 122–133.

[13] Sridevi, G.; Kumar, S.S. p-Laplace Variational Image Inpainting Model Using Riesz Fractional Differential Filter. *Int. J. Electr. Comput. Eng.* 2017, *7*, 850–857.

[14] Sridevi, G.; Kumar, S.S. Image Inpainting and Enhancement using Fractional Order Variational Model. *Def. Sci. J.* 2017, *67*, 308–315.

[15] Sridevi, G.; Kumar, S.S. Image Inpainting Based on Fractional-Order Nonlinear Diffusion for Image Reconstruction. *Circuits Syst. Signal Process.* 2019, *38*, 3802–3817.

[16] Gamini, S.; Gudla, V.V.; Bindu, C.H. Fractional-order Diffusion based Image Denoising Mode. *Int. J. Electr. Electron. Res.* 2022, *10*, 837–842.

[17] Papafitsoros, K.; Schoenlieb, C.B.; Sengul, B. Combined First and Second Order Total Variation Inpainting using Split Bregman. *Image Process. Line* 2013, *3*, 112–136.

[18] Efros, A.A.; Leung, T.K. Texture Synthesis by Non-parametric Sampling. In Proceedings of the Seventh IEEE International Conference on Computer Vision, Kerkyra, Greece, 20–27 September 1999.

[19] Criminisi, A.; Pérez, P.; Toyama, K. Region filling and object removal by exemplar-based image inpainting. *IEEE Trans. Image Process.* 2004, *13*, 1200–1212.

[20] Ružić, T.; Pižurica, A. Context-aware patch-based image inpainting using Markov random field modeling. *IEEE Trans. Image Process.* 2015, *24*, 444–456.

[21] Jin, K.H.; Ye, J.C. Annihilating Filter-Based Low-Rank Hankel Matrix Approach for Image Inpainting. *IEEE Trans. Image Process.* 2015, *24*, 3498–3511.

[22] Kawai, N.; Sato, T.; Yokoya, N. Diminished Reality Based on Image Inpainting Considering Background Geometry. *IEEE Trans. Vis. Comput. Graph.* 2016, *22*, 1236–1247.

[23] Guo, Q.; Gao, S.; Zhang, X.; Yin, Y.; Zhang, C. Patch-Based Image Inpainting via Two-Stage Low Rank Approximation. *IEEE Trans. Vis. Comput. Graph.* 2018, *24*, 2023–2036.

[24] Lu, H.; Liu, Q.; Zhang, M.; Wang, Y.; Deng, X. Gradient-based low rank method and its application in image inpainting. *Multimed. Tools Appl.* 2018, *77*, 5969–5993.

[25] Shen, L.; Xu, Y.; Zeng, X. Wavelet inpainting with the ℓ0 sparse regularization. *Appl. Comput. Harmon. Anal.* 2016, *41*, 26–53.

[26] Waller, B.M.; Nixon, M.S.; Carter, J.N. Image reconstruction from local binary patterns. In Proceedings of the 2013 International Conference on Signal-Image Technology & Internet-Based Systems, Kyoto, Japan, 2–5 December 2013; pp. 118–123.

[27] Li, H.A.; Hu, L.; Liu, J.; Zhang, J.; Ma, T. A review of advances in image inpainting research. *Imaging Sci. J.* 2023.

[28] Rasaily, D.; Dutta, M. Comparative theory on image inpainting: A descriptive review. In Proceedings of the 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS), Chennai, India, 1–2 August 2017; pp. 2925–2930.

[29] Shen, B.; Hu, W.; Zhang, Y.; Zhang, Y.J. Image inpainting via sparse representation. In Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing, Taipei, China, 19–24 April 2009; pp. 697–700.

[30] Xu, Z.; Sun, J. Image inpainting by patch propagation using patch sparsity. *IEEE Trans. Image Process.* 2010, *19*, 1153–1165.

1. INTRODUCTION

2.LITERATURE SURVEY

3.EXISTING SYSTEM

4.PROPOSED SYSTEM

5.UML DIAGRAMS

6.SOFTWARE ENVIRONMENT

7.FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

8. SOURCE CODE

9. RESULTS AND DISCUSSION

10. CONCLUSION AND FUTURE SCOPE