Image Stitching
Final Project Report of CS/EIE/SE 460

Huo, 1210022986; Liang, 1210021720

December 2024
Shool of Computer Science and Engineering, Macau University of Science and Technology

1 Introduction

1.1 Task Description

The task at hand is to create a panoramic image by stitching multiple overlapping images into one continuous image. This process, known as image stitching, involves aligning a set of images and merging them in such a way that the transition between individual images is seamless.

1.2 Motivation

Image stitching has a wide range of applications, from creating high-resolution panoramas for virtual tours and photography to generating composite images for medical imaging or satellite imagery analysis. The motivation behind this project is to develop an efficient and automated system that can handle the challenges associated with image stitching, such as varying lighting conditions, perspective changes, and misalignments due to camera movement.

1.3 Difficulties

- Detecting and matching features across different images accurately.
- Handling changes in scale, rotation, and perspective.
- Dealing with non-uniform illumination and color differences between images.
- Blending images together seamlessly without visible seams or artifacts.

1.4 Solution Overview

To address these challenges, we utilize Scale-Invariant Feature Transform (SIFT) for feature detection and matching, FLANN-based matcher for finding correspondences between images, and RANSAC algorithm for robust estimation of homography matrix which defines the geometric transformation between two images. The blending of images is done using a weighted average method in the overlapping regions to ensure a smooth transition between adjacent images.

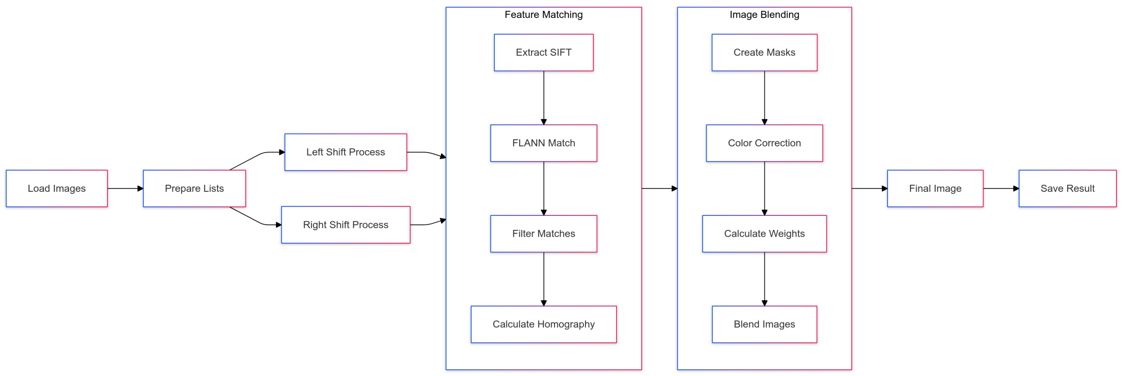


Figure 1: Flow Chart

2 Technical Solution

The overall framework consists of several key components:

1) **Feature Detection and Matching**: Extracts distinctive features from images using SIFT and matches them using FLANN.
2) **Homography Estimation**: Estimates the homography matrix using RANSAC on matched feature points.
3) **Image Transformation**: Applies the homography matrix to warp images into a common coordinate system.
4) **Blending**: Blends overlapping regions of adjacent images to create a seamless panorama.
5) **Stitching Execution**: Sequentially stitches images from a central image towards both sides.

2.1 Conceptual Introduction

- **Keypoint Detection**: Keypoint detection is like looking for "unique landmarks" in a picture, for example, if you want to describe a city photo, you might mention "that tower" or "that red sign" because they are easily recognizable. For example, if you want to describe a city photo, you might mention "that tower" or "that red sign" because these landmarks are unique and easily recognizable. What the SIFT algorithm does is to find these "landmarks" in the image and ensure that they are recognized whether the image is zoomed in, zoomed out, or rotated. SIFT (Scale-Invariant Feature Transform) is a robust algorithm used to detect and describe local features in images. It identifies keypoints that are invariant to image scale and rotation, and partially invariant to illumination changes and affine transformations. The SIFT algorithm generates a 128-dimensional descriptor vector for each keypoint, which can be used to match features across different images reliably. This makes SIFT an ideal choice for tasks requiring stable and repeatable feature detection, such as image stitching.
  SIFT extracts keypoints by detecting extreme points (in scale and space) in an image and generates descriptors. Its main steps include:

  1) **Scale Spatial Extreme Value Detection**:
     - Construct scale space using Gaussian pyramid to detect extreme points.
     - Scale space definition:

     $$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

     where $G(x, y, \sigma)$ is the Gaussian kernel, $I(x, y)$ is the input image, and σ is the scale parameter. - Extreme points are detected by Differential Gaussian (DoG):

     $$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

  2) **Key point localization**: Reject low contrast and edge response points by Taylor expansion and Hessian matrix.
  3) **Direction assignment**: Calculate the principal orientation of keypoints to make the descriptor rotation invariant.
  4) **Descriptor generation**: Compute the gradient histogram in the region around the keypoint to generate a 128-dimensional descriptor.

- **Feature Matching: FLANN (Fast Library for Approximate Nearest Neighbors)**: Feature matching is like playing "find the same". Suppose you have two photos, each with some unique landmarks (keypoints), and your task is to match the landmarks that are the same in both photos. FLANN is designed to find these pairs quickly, rather than comparing all the possible combinations one by one, thus saving you a lot of time. More specifically, it is designed to perform fast approximate nearest neighbor searches in high-dimensional spaces. It supports multiple algorithms, including KD-trees, which are particularly efficient for SIFT descriptors due to their dimensionality. By using FLANN, we can efficiently find the best matches between keypoints detected by SIFT, significantly speeding up the matching process while maintaining accuracy. This combination of SIFT for feature extraction and FLANN for matching provided robust and precise keypoint correspondences essential for successful stitching.
  FLANN is an efficient Approximate Nearest Neighbor search algorithm that accelerates matching by constructing KD trees or K-means trees.

1) **KD tree construction**: Divide the feature description subspace into multiple hyper-rectangular regions to construct the tree structure.
2) **Nearest Neighbor Search**: Reject low contrast and edge response points by Taylor expansion and Hessian matrix.
3) **Match Screening**: Rejects false matches using Ratio Test:

$$\frac{d_1}{d_2} < \text{threshold}$$

Where d_1 and d_2 are the distances to the nearest neighbor and next nearest neighbor respectively.

- **Geometric Transformation Estimation: RANSAC (Random Sample Consensus)**: Geometric transformation estimation is like adjusting the position of two photos so that they can be perfectly aligned. For example, if you want to stitch together two photos taken at slightly different angles, you need to know how to rotate, scale, or move one of the photos to match the other, and RANSAC is there to eliminate any "messed up" mismatches in the process, ensuring that the final alignment is accurate.RANSAC estimates transformation matrices (e.g., single-stress matrices) by random sampling and iterative optimization, and filters out "interior points" that match the model. This method is robust to noise and mismatches, and can effectively improve the accuracy of geometric transformations.
RANSAC is a robust parameter estimation method for rejecting false matches and estimating the transform matrix.

1) **Random Sample**: Randomly select the smallest sample set (e.g., 4 pairs of points) from the matched pairs.
2) **Model estimation**: Calculate the transformation matrix H using the minimum sample set (e.g., single-stress matrix):

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Where (x, y) and (x', y') are matching point pairs.
3) **Internal Point Screening**: Calculate the projection error of all matched point pairs and filter the inner points:

$$|p' - Hp\ | < \text{threshold}$$

4) **Iterative Optimization**: Repeat the above process and select the model with the most inner points as the final result.

- **Image Blending**: Image fusion is like "stitching" two photos together and making the seams look natural and seamless. For example, if you stitch together two photos of the same scene, there may be inconsistencies in color or brightness in the overlapping areas. Image fusion technology makes these transitions look smooth and natural through methods such as weighted averaging or multi-band fusion. Weighted average fusion makes the transition smoother by weighted averaging the pixels of the two images in the overlapping region. Multi-band fusion, on the other hand, avoids artifacts at the seams by decomposing the image into bands of different frequencies and fusing them separately before reconstruction.
Image blending is the seamless stitching of aligned images, and commonly used methods include weighted average and multi-band fusion.

1) **Weighted Average Fusion**: In the overlapping region, the pixels of the two images are weighted and averaged:

$$I_{\text{blend}}(x, y) = w_1 I_1(x, y) + w_2 I_2(x, y)$$

Where w_1 and w_2 are the weights, which are usually determined based on the distance from the pixel to the boundary.
2) **Multi-band fusion**: Decompose the image into multiple frequency bands and fuse them separately before reconstruction:

$$I_{\text{blend}} = \sum_k \text{Blend}(I_{1,k}, I_{2,k})$$

Where $I_{1,k}$ and $I_{2,k}$ are the first k bands of the two images.

2.2 Feature Detection and Matching

- **Initialization (*__init__*)**: Initializes the SIFT detector and sets up FLANN parameters. FLANN is used because it is fast and can efficiently find approximate nearest neighbors in high-dimensional spaces, which is ideal for SIFT descriptors. The trees parameter specifies the number of randomized trees to be used for indexing, while checks defines the number of times the tree is recursively traversed.

- ***getSIFTFeatures* Function**: Converts the input image to grayscale and uses the SIFT algorithm to detect keypoints and compute descriptors. Keypoints are locations in the image that are invariant to scale and rotation, making them robust to changes in viewpoint. Descriptors provide a unique representation of the area around each keypoint, allowing for matching between images. The formula for SIFT involves detecting extrema in the Difference of Gaussians (DoG) scale-space, followed by orientation assignment and descriptor generation.

```
class Matchers:
    """
    A class to handle feature matching between images using SIFT and FLANN
    """
    def __init__(self):
        # Initialize SIFT detector
        self.sift = cv2.SIFT_create()
        # FLANN parameters for SIFT matching
        FLANN_INDEX_KDTREE = 1  # Using FLANN_INDEX_KDTREE for SIFT
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)
        self.flann = cv2.FlannBasedMatcher(index_params, search_params)

    def getSIFTFeatures(self, im):
        """
        Extract SIFT features from an image
        Args:
            im: Input image
        Returns:
            Dictionary containing keypoints and descriptors
        """
        gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
        kp, des = self.sift.detectAndCompute(gray, None)
        return {'kp': kp, 'des': des}
```

2.3 Homography Estimation

- ***match* Function**: Finds corresponding features between two images by applying a ratio test to filter out ambiguous matches. The ratio test compares the distance of the best match to the second-best match for each keypoint; only when the best match is significantly better is it considered reliable. Then it estimates the homography matrix using RANSAC. RANSAC works by randomly selecting subsets of data points (inliers), estimating the model parameters from these points, and evaluating how many other points fit this model within a certain threshold. The homography matrix H' satisfies the equation $p' = Hp$ where p and p' are corresponding points in the two images.

```
def match(self, img1, img2, direction=None):
        """
        Match features between two images
        Args:
            img1, img2: Input images to be matched
            direction: Stitching direction (left/right)
        Returns:
            H: Homography matrix for image transformation
        """
        # Extract SIFT features from both images
        imageSet1 = self.getSIFTFeatures(img1)
        imageSet2 = self.getSIFTFeatures(img2)
        print("Direction:", direction)

        # Find k-nearest matches for each descriptor
        matches = self.flann.knnMatch(imageSet2['des'], imageSet1['des'], k=2)
        good = []
        # Apply ratio test to filter good matches
        for m, n in matches:
            if m.distance < 0.6 * n.distance:
                good.append((m.trainIdx, m.queryIdx))

        # Calculate homography if enough good matches are found
        if len(good) > 4:
            pointsCurrent = imageSet2['kp']
            pointsPrevious = imageSet1['kp']

            matchedPointsCurrent = np.float32([pointsCurrent[i].pt for (_, i) in good])
            matchedPointsPrev = np.float32([pointsPrevious[i].pt for (i, _) in good])

            # Find homography using RANSAC
            H, _ = cv2.findHomography(matchedPointsCurrent, matchedPointsPrev, cv2.RANSAC, 4)
            return H
        return None
```

2.4 Image Transformation

- **Initialization (*__init__*)**: Loads images from a specified folder, resizes them, and prepares lists of images to be stitched to the left and right of a central image. It also initializes the Matchers object for feature matching.
- ***load˙images* Function**: Loads images from the given folder path and resizes them to a fixed size. Resizing helps to reduce computational complexity during processing. The function filters files by extension to ensure only images are processed.
- ***prepare_lists* Function**: Prepares the list of images for stitching. It divides the images into left and right lists based on the central image. This central image serves as the anchor point for stitching, reducing cumulative errors as images are aligned sequentially.

```
class Stitch:
    """
    Main class for image stitching operations
    """
    def __init__(self, image_folder):
        """
        Initialize the stitching process
        Args:
            image_folder: Path to folder containing images to be stitched
        """
        self.images = self.load_images(image_folder)
        self.count = len(self.images)
        self.left_list, self.right_list, self.center_im = [], [], None
        self.matcher_obj = Matchers()
        self.prepare_lists()

    def load_images(self, folder):
        """
        Load and resize all images from the specified folder
        Args:
            folder: Path to image folder
        Returns:
            List of resized images
        """
        filenames = [f for f in os.listdir(folder) if f.endswith(('jpg', 'jpeg', 'png', 'JPG'))]
        images = [cv2.resize(cv2.imread(os.path.join(folder, f)), (480, 320)) for f in filenames]
        return images

    def prepare_lists(self):
        """
        Prepare image lists for left and right stitching
        Divides images into left and right lists based on center image
        """
        print(f"Number of images: {self.count}")
        self.centerIdx = self.count // 2
        print(f"Center index image: {self.centerIdx}")
        self.center_im = self.images[self.centerIdx]
        self.left_list = self.images[:self.centerIdx + 1]
        self.right_list = self.images[self.centerIdx + 1:]
        print("Image lists prepared")
```

- ***leftshift* Function**:
  1) **Stitching Process**: Starts with the leftmost image and iteratively applies homography matrices to align subsequent images. The inverse homography matrix is calculated to transform the current image to the coordinate system of the previous one. Offsets are calculated to ensure no part of the image is cut off due to the transformation.
  2) **Perspective Transformation**: Uses OpenCV's warpPerspective function to apply the homography matrix H to the image. The function warps the source image using the specified matrix, producing a new image with the correct perspective. The dsize parameter defines the size of the output image, which includes any necessary offsets to accommodate the entire transformed image.

```
def leftshift(self):
        """
        Stitch images to the left of the center image
        Applies perspective transformation and combines images
        """
        # Start with the leftmost image in the left list
        a = self.left_list[0]

        # Iterate through remaining images from left to center
        for b in self.left_list[1:]:
            # Find homography matrix between consecutive images
            H = self.matcher_obj.match(a, b, 'left')
            print("Homography is:", H)

            # Calculate inverse homography matrix for perspective transform
            xh = np.linalg.inv(H)

            # Calculate dimensions of warped image using corner points
            ds = np.dot(xh, np.array([a.shape[1], a.shape[0], 1]))
            ds = ds / ds[-1]  # Normalize homogeneous coordinates

            # Calculate offset for the origin point (0,0)
            f1 = np.dot(xh, np.array([0, 0, 1]))

            # Adjust homography matrix to handle negative offsets
            xh[0][-1] += abs(f1[0])  # Add x-offset to transformation
            xh[1][-1] += abs(f1[1])  # Add y-offset to transformation

            # Recalculate final image dimensions after offset adjustment
            ds = np.dot(xh, np.array([a.shape[1], a.shape[0], 1]))

            # Calculate absolute offsets for image placement
            offsety = abs(int(f1[1]))  # Vertical offset
            offsetx = abs(int(f1[0]))  # Horizontal offset

            # Define size of output image including offsets
            dsize = (int(ds[0]) + offsetx, int(ds[1]) + offsety)
            print("Image dsize =>", dsize)

            # Apply perspective transform to first image
            tmp = cv2.warpPerspective(a, xh, dsize)

            # Copy second image onto the transformed first image
            tmp[offsety:b.shape[0] + offsety, offsetx:b.shape[1] + offsetx] = b
            # Update working image for next iteration
            a = tmp

        # Store final left-stitched image
        self.leftImage = tmp
```

- ***rightshift* Function**:
  1) **Stitching Process**: Similar to leftshift, but processes images to the right of the central image. It finds the homography matrix between the currently accumulated left image and the next right image to be stitched.
  2) **Transformation and Blending**: After calculating the homography matrix, it applies the transformation to the right image. A new canvas is created for the combined image, where the existing left image is copied. The overlapping regions between the transformed right image and the left image are blended using the mix_and_match function.
  3) **Valid Region Identification**: After blending, the function creates a binary mask to identify the valid region of the stitched image. Contours are found, and the largest one is used to crop the final image, removing any black borders caused by the transformation.

```
def rightshift(self):
        """
        Stitch images to the right of the center image
        Applies perspective transformation and combines images with blending
        """
        # Process each image in right list
        for each in self.right_list:
            # Find homography matrix between left image and current right image
            H = self.matcher_obj.match(self.leftImage, each, 'right')
            print("Homography:", H)

            # Get dimensions of current left image
            h, w = self.leftImage.shape[:2]

            # Calculate new image dimensions after transformation
            txyz = np.dot(H, np.array([each.shape[1], each.shape[0], 1]))
            txyz = txyz / txyz[-1]  # Normalize homogeneous coordinates

            # Calculate size of new canvas needed
            dsize = (int(txyz[0]) + self.leftImage.shape[1], int(txyz[1]) + self.leftImage.shape[0])

            # Apply perspective transform to right image
            tmp = cv2.warpPerspective(each, H, dsize)

            # Create new blank canvas for composite image
            new_img = np.zeros((dsize[1], dsize[0], 3), dtype=np.uint8)

            # Copy left image to new canvas
            new_img[:h, :w] = self.leftImage

            # Blend the overlapping regions
            result = self.mix_and_match(new_img, tmp)
```

```
    # Find valid region in stitched image by creating binary mask
    gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)

    # Find contours of valid image region
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Crop to valid region if contours found
    if contours:
        # Find bounding rectangle of largest contour
        x, y, w, h = cv2.boundingRect(max(contours, key=cv2.contourArea))
        self.leftImage = result[y:y+h, x:x+w]
    else:
        self.leftImage = result
```

2.5 Blending

To ensure smooth transitions at overlap regions, a blending function `mix_and_match()` calculates weights for each pixel in the overlap area based on its distance from the edges:

$w(x) = \frac{x - left\_edge}{right\_edge - left\_edge}$

A Gaussian blur is then applied to these weights to create a smoother transition:

$w'(x) = G_\sigma(w(x))$

This ensures that pixels closer to the left boundary take on colors more similar to those in the left image, while pixels near the right boundary resemble the right image more closely.

- **Blending Process**: Creates a smooth transition between the overlapping regions of two images. It starts by converting the images to grayscale and creating binary masks. These masks are used to find the overlapping region, which is then dilated to expand the area where blending will occur.
- **Weight Map Creation**: Constructs a weight map that assigns weights to pixels in the overlapping region based on their distance from the edge of the overlap. Pixels closer to the left edge receive a lower weight for the right image and vice versa. This creates a gradual blend between the two images.
- **Applying Weights**: Multiplies the pixel values of the two images by their respective weights and sums the results to obtain the blended image. Non-overlapping regions are copied directly from the original images to avoid artifacts.

```
def mix_and_match(self, leftImage, warpedImage):
        """
        Blend two images in their overlapping region
        Args:
            leftImage: Left side image
            warpedImage: Transformed right side image
        Returns:
            result: Blended image
        """
        # Create masks for overlap region
        gray_left = cv2.cvtColor(leftImage, cv2.COLOR_BGR2GRAY)
        gray_right = cv2.cvtColor(warpedImage, cv2.COLOR_BGR2GRAY)

        # Create binary masks
        _, mask_left = cv2.threshold(gray_left, 1, 255, cv2.THRESH_BINARY)
        _, mask_right = cv2.threshold(gray_right, 1, 255, cv2.THRESH_BINARY)
```

```
# Find overlapping region
overlap = cv2.bitwise_and(mask_left, mask_right)

# Dilate overlap region
kernel_size = 100  # Increase kernel size for smoother transition
kernel = np.ones((kernel_size,kernel_size), np.uint8)
overlap_dilated = cv2.dilate(overlap, kernel)

# Create weight map
rows, cols = overlap_dilated.shape
distances = np.zeros((rows, cols))

# Calculate distances to edges
for y in range(rows):
    for x in range(cols):
        if overlap_dilated[y,x] > 0:
            left_edge = x
            right_edge = x
            for i in range(x, -1, -1):
                if overlap_dilated[y,i] == 0:
                    left_edge = i
                    break
            for i in range(x, cols):
                if overlap_dilated[y,i] == 0:
                    right_edge = i
                    break
            # Calculate relative distance as weight
            if right_edge > left_edge:
                distances[y,x] = float(x - left_edge) / float(right_edge - left_edge)

# Smooth weight map
distances = cv2.GaussianBlur(distances, (51,51), 0)

# Create three-channel weights
weight_right = cv2.merge([distances, distances, distances])
weight_left = 1 - weight_right

# Apply weights
result = leftImage.astype(float) * weight_left + warpedImage.astype(float) * weight_right

# Copy non-overlapping regions directly
result[mask_left == 0] = warpedImage[mask_left == 0]
result[mask_right == 0] = leftImage[mask_right == 0]

return result.astype(np.uint8)
```

2.6 Stitching Execution

Beyond core functionalities, the system provides a user-friendly interface that allows users to initiate the stitching process via command-line tools and supports saving or displaying final results.

- ***showImage* Function**: Displays the final stitched panorama using OpenCV's window display functions. It shows the image until a key is pressed, after which the window is closed.

```
def showImage(self):
        """
        Display the final stitched image
        """
        cv2.imshow("Stitched Image", self.leftImage)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

- ***main* Function**: The main function initializes the Stitch object with the path to the folder containing the images to be stitched. It calls leftshift to stitch images to the left of the central image, then rightshift to stitch images to the right. Finally, it displays the stitched panorama and saves it as a JPEG file.

```
if __name__ == '__main__':
    # Specify the folder containing images to be stitched
    image_folder = 'street'
    stitcher = Stitch(image_folder)
    stitcher.leftshift()
    stitcher.rightshift()
    stitcher.showImage()
    cv2.imwrite("stitched_street.jpg", stitcher.leftImage)
    print("Stitched image saved as stitched_output.jpg")
```

3 Experiments

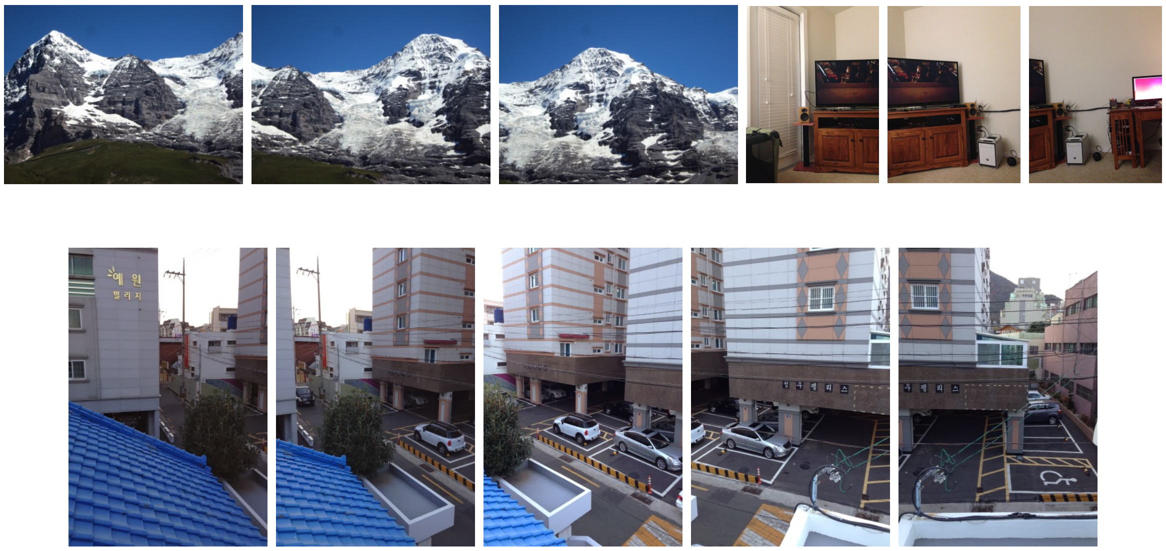


Figure 2: Input Image Groups: mountain, room, street

To evaluate the performance of our image stitching system, we conducted experiments using a dataset of street scene photos. Each image was resized to a uniform dimension of 480x320 pixels for consistency. The system successfully handled various challenges, including significant viewpoint differences and changes in lighting conditions. Additionally, GPU acceleration significantly sped up the entire processing pipeline, making the system viable for real-time applications.

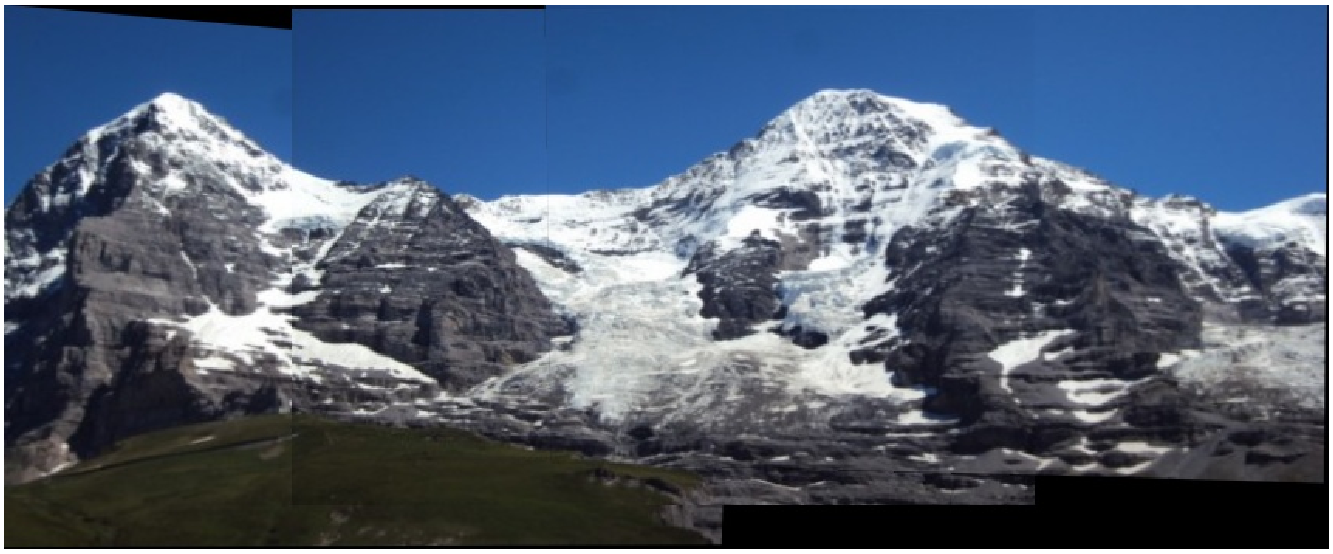


Figure 3: Stitched Mountain

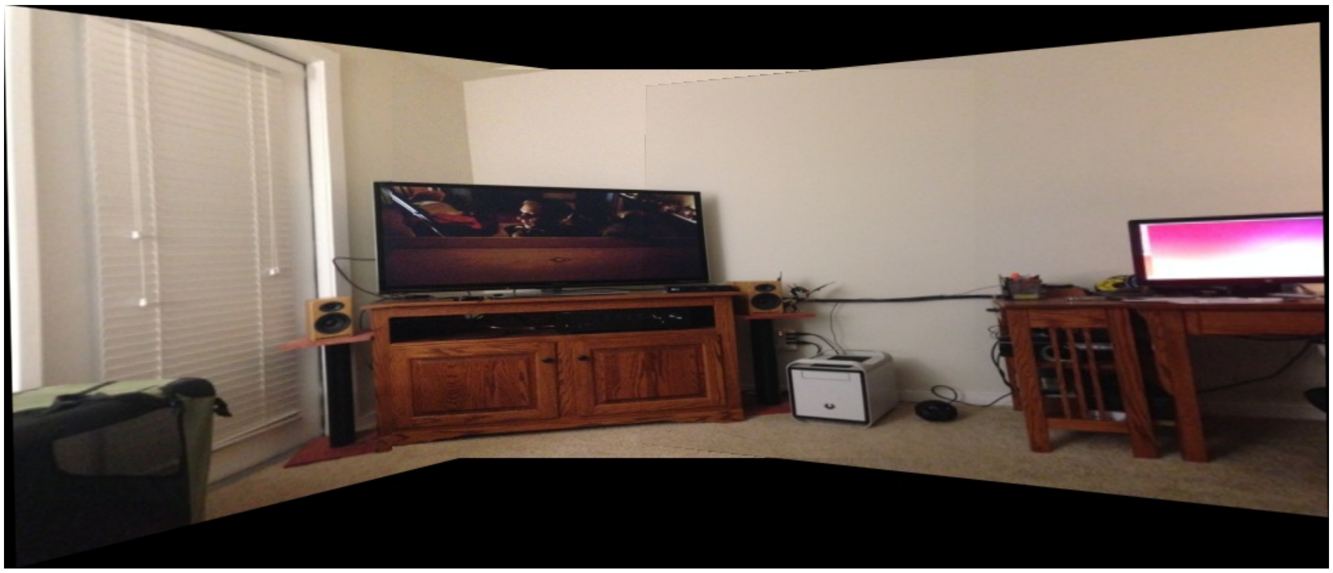


Figure 4: Stitched Room

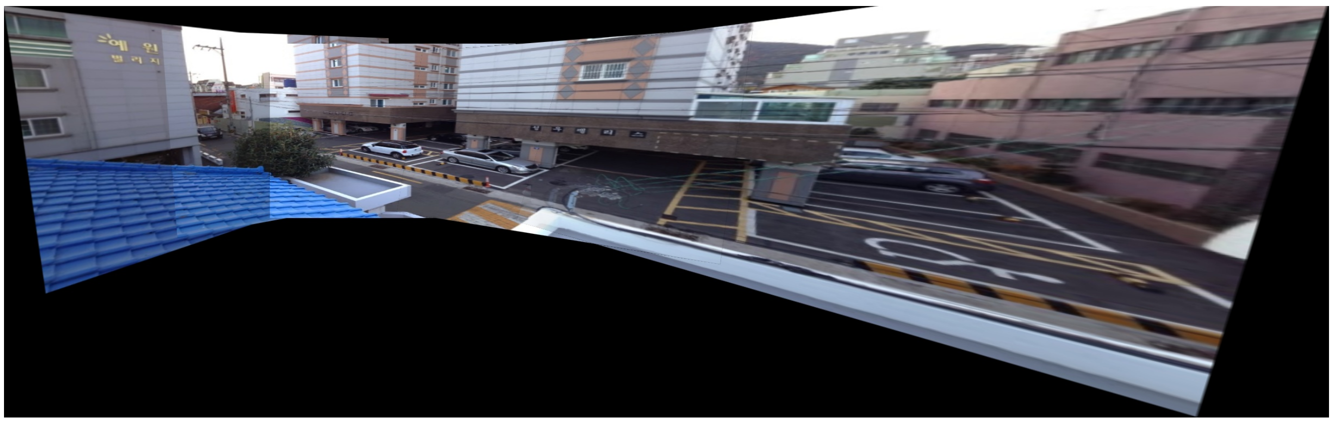
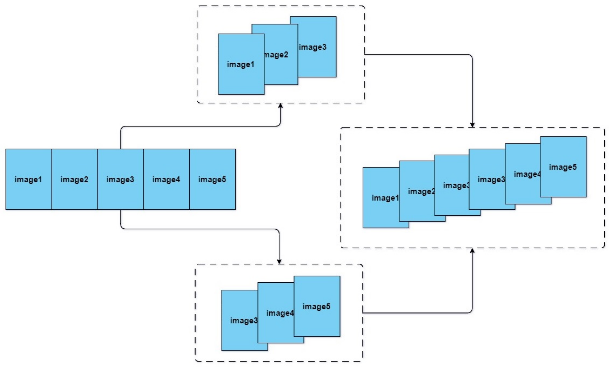


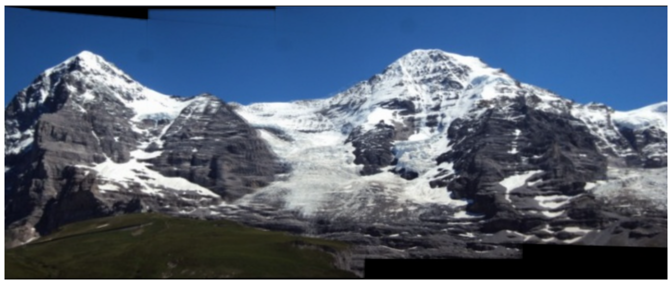
Figure 5: Stitched Street

4 Discussion

In our research, we developed a unique image stitching scheme that expands from a central image in two directions to generate two separate images – one to the left and one to the right – which are ultimately merged into a single panoramic image. To ensure accurate keypoint correspondence and proper image alignment, we employed the SIFT (Scale-Invariant Feature Transform) algorithm for feature extraction and paired it with FLANN (Fast Library for Approximate Nearest Neighbors) matching. This combination provided robust and precise keypoint matches. Furthermore, homography estimation and perspective transformations enabled us to align images accurately, even under challenging conditions. However, during implementation, we encountered several challenges.



First, when dealing with extreme viewpoint changes or significant occlusions, the performance of the matching and transformation algorithms was somewhat limited, indicating the need for further optimization to enhance their robustness and accuracy. Second, we observed that keypoints in the stretched areas on either side of the images affected the final stitching of the two images, causing the central region, which should have remained distortion-free, to also be stretched. This issue suggests that while SIFT and FLANN offer high-quality keypoint matching, they require complementary methods to maintain the true scale of the central area within our specific stitching process. Regarding the noticeable boundary lines after stitching, we adopted Gaussian blurring as a solution, applying this



technique only to a narrow region around the seam. This approach effectively reduced the visibility of the boundary line but also highlighted a trade-off: if the blurred area is too small, it does not sufficiently mask the stitching mark; if the blur region is expanded, it can lead to unnecessary blurring. Thus, achieving a smooth transition along the boundary line while preserving image details remains an ongoing challenge.

Looking forward, we will consider incorporating more advanced image blending techniques, such as deep learning-based methods, and exploring new stitching strategies to address the limitations of current methods, especially in handling extreme viewpoint changes and large occlusions. We will also continue to refine the existing matching and transformation algorithms to achieve better stitching outcomes.

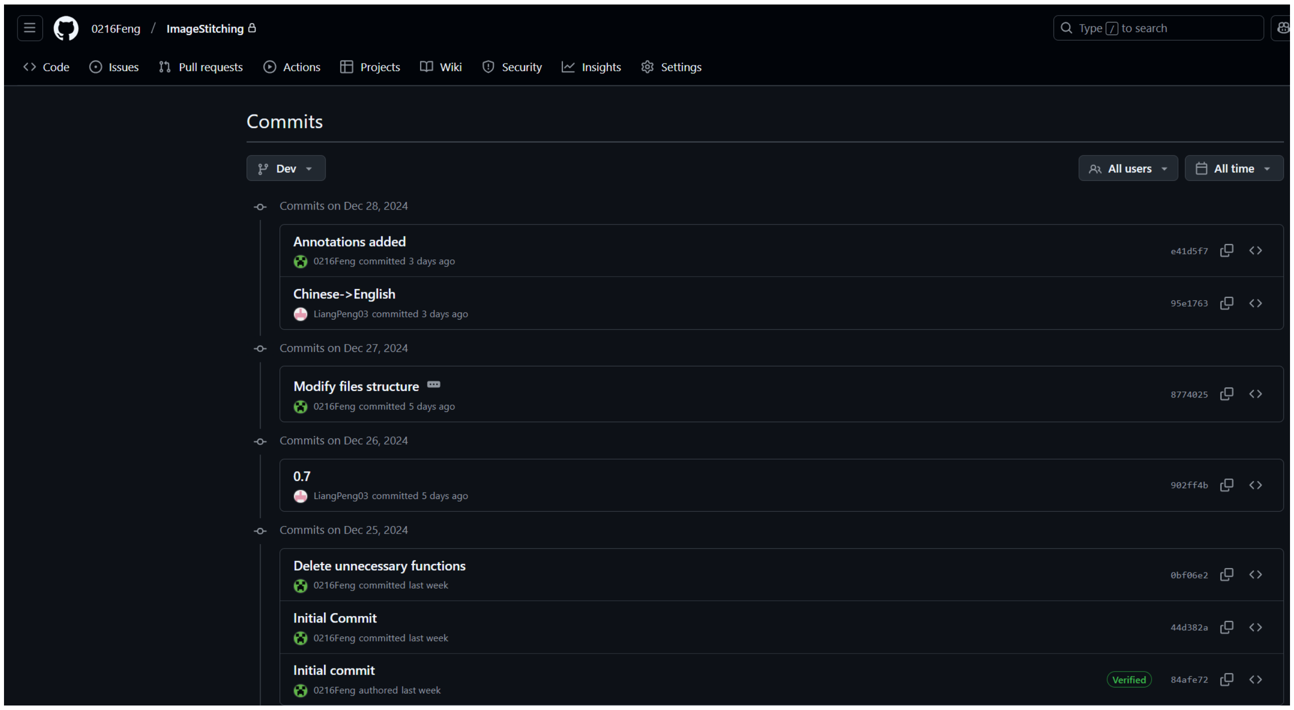
In conclusion, while our current approach delivers satisfactory results in most cases, there is room for improvement. Through continuous innovation and methodological refinement, we aim to achieve more optimal results in future research, contributing valuable advancements to the field of image stitching.

5 Contribution Description

The contribution of the project of our group members will be discussed in terms of development and documentation.

5.1 Development

- **Coding**:
    1. Huo Zhifeng: Implemented the initial version and make necessary modifications; Added necessary comments in code for legibility; Modified the intermediate version done by Liang Peng and improved the displaying results by making the images split line less obvious.
    2. Liang Peng: Collected reference document for developing; Provided ideas of improvement; Modified the initial version and solved the picture perspective problem; Added more image groups for tesing; Code review.
- **GitHub Repository Maintenance**:
    1. Huo Zhifeng: Created repository; Committed new versions and modified files structure to ensure well maintainability and availability.
    2. Liang Peng: Committed new versions and made necessary changes; Assisted in maintaining the repository.



Commits

Commits on Dec 28, 2024

Annotations added
0216Feng committed 3 days ago — e41d5f7

Chinese->English
LiangPeng03 committed 3 days ago — 95e1763

Commits on Dec 27, 2024

Modify files structure
0216Feng committed 5 days ago — 8774025

Commits on Dec 26, 2024

0.7
LiangPeng03 committed 5 days ago — 902ff4b

Commits on Dec 25, 2024

Delete unnecessary functions
0216Feng committed last week — 0bf06e2

Initial Commit
0216Feng committed last week — 44d382a

Initial commit
0216Feng authored last week — Verified — 84afe72

Figure 6: GitHub Commit Record

5.2 Documentation

1. Huo Zhifeng: Reviewed and Modified the Introduction and Technical Solution parts based on the initial version of Liang Peng; Add some styles for code and paragraghs to ensure aesthetic and readability; Wrote Contribution part; Layout modification; Final review of document.
2. Liang Peng: Wrote initial version of Introduction and Technical Solution parts; Wrote in-depth analysis in Experiments and Discussion parts with detailed pictures based on the full knowledge of project; Wrote reference in standard citation format; Layout modification; Final review of document.

6 References

[1] Opencv Practice - Image Stitching.(2022, June 6). CSDN. https://blog.csdn.net/Thousand_drive/article/details/125084810.
[2] Computer Vision: Panorama Stitching.(2022, December 19). CSDN. https://blog.csdn.net/weixin_43603658/article/details/128373038.
[3] Lowe, D. G. (1999). Object recognition from local scale-invariant features. In Proceedings of the Seventh IEEE International Conference on Automatic Face and Gesture Recognition (pp. 1150-1157). IEEE.
[4] Suju, D. A., Jose, H. (2017). FLANN: Fast approximate nearest-neighbor search algorithm for elucidating human-wildlife conflicts in forest areas. IEEE. https://doi.org/10.1109/icscn.2017.8085676.