Homework 2 – Panorama Stitching Report

Author: Xinjie Chen

Date: September 29, 2025

1. A brief description of the programs

In this homework, I implemented a minimal panorama stitching pipeline without using OpenCV's cv.Stitcher. The program follows the workflow described in class and lecture slides:

- First, SIFT was used to detect local features and compute descriptors in each input image. The number of detected keypoints was also reported.
- Next, pairwise feature matching between consecutive images was performed using BFMatcher and Lowe's ratio test. Before applying RANSAC, the top-10 scoring matches were visualized.
- RANSAC was then used to estimate a homography for each image pair. The number of inliers was printed, and the homography matrix was saved. The top-10 inliers with the smallest reprojection error were also visualized.
- After that, the middle image was chosen as the anchor. Homographies from all other images were composed to this anchor, and a global canvas bounding box was computed. Each image was warped into this common canvas.
- Finally, the three images were warped into the same canvas and blended together to generate the final panorama.

2. Results of intermediate steps

For this assignment, I captured three photos casually in the back alley near my apartment. The images had sufficient overlap of the building corner and power poles, making them suitable for panorama stitching.

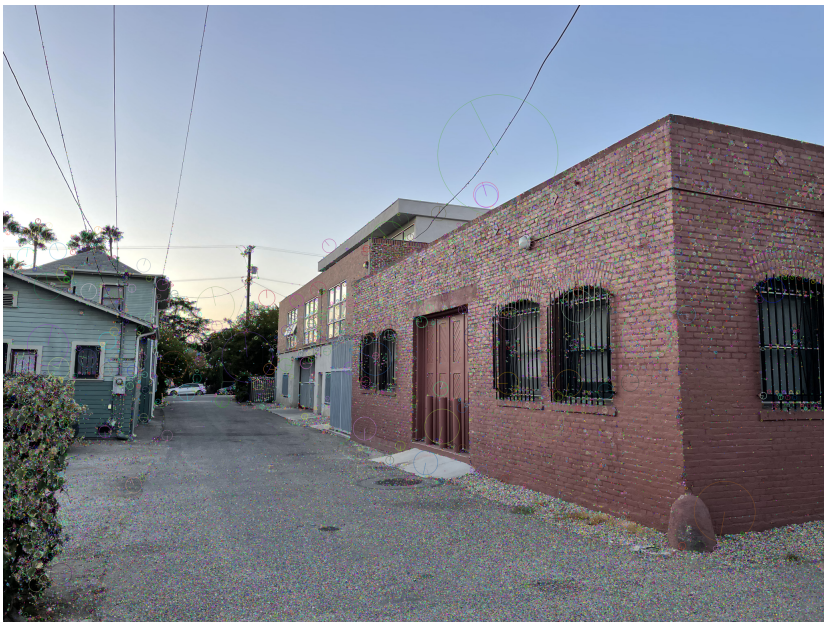
- **SIFT features**: Thousands of features were detected per image, mainly concentrated on the brick wall edges, windows, and utility poles.

- **Pre-RANSAC matches**: The top-10 matches were visualized. They mostly aligned around window corners and pole intersections.
- **Post-RANSAC matches**: The top-10 inliers with the smallest reprojection error showed very good alignment around strong edges.
- **Warped images**: Each input was successfully warped into the canvas. Some areas appeared distorted at the edges, which is expected with perspective transformation.
- **Final panorama**: The three alley images were stitched into a single wide-angle view. The building corner aligned reasonably well, though some ghosting can still be observed in the overlapping region near the wires and a small part of the brick wall.

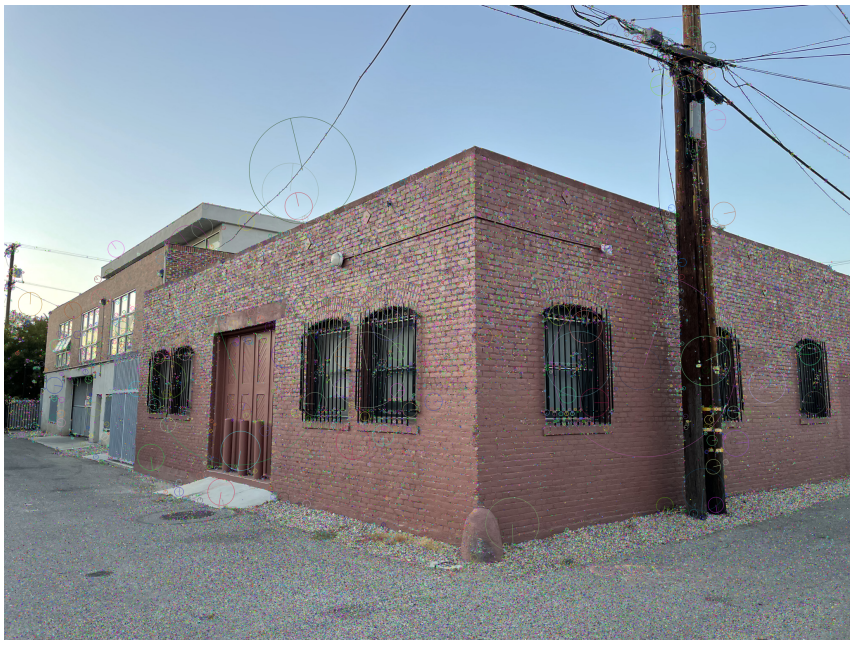
3. Observations

- **Ratio test sensitivity**: Adjusting the ratio from 0.75 to 0.70 made the matches stricter, which slightly reduced ghosting in my final panorama.
- **Homography results**: RANSAC was able to find a large number of inliers, which confirmed that the images were well-suited for homography-based stitching. However, because the alley had repeating patterns (bricks and windows), some mismatches still occurred.
- **Final panorama quality**: The panorama clearly captured the continuous alley scene, including the long red-brick wall and power poles. However, ghosting artifacts were noticeable on moving objects (like leaves) and repetitive textures.
- **Practical lesson**: I realized that even when using a robust pipeline, real-world casual photos (taken quickly in the alley without a tripod) will often result in imperfect panoramas. Small camera rotations or parallax from moving sideways can cause noticeable distortions.

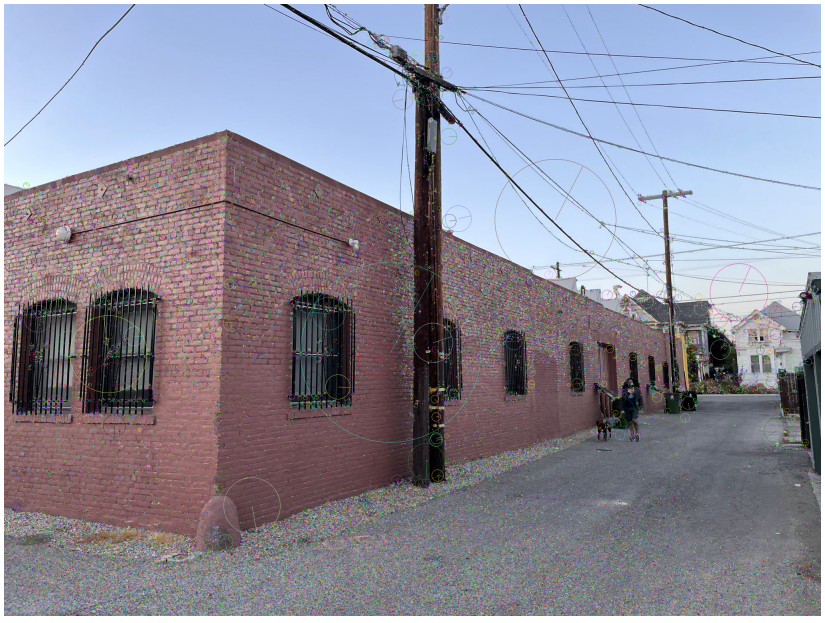
SIFT Features Image 1



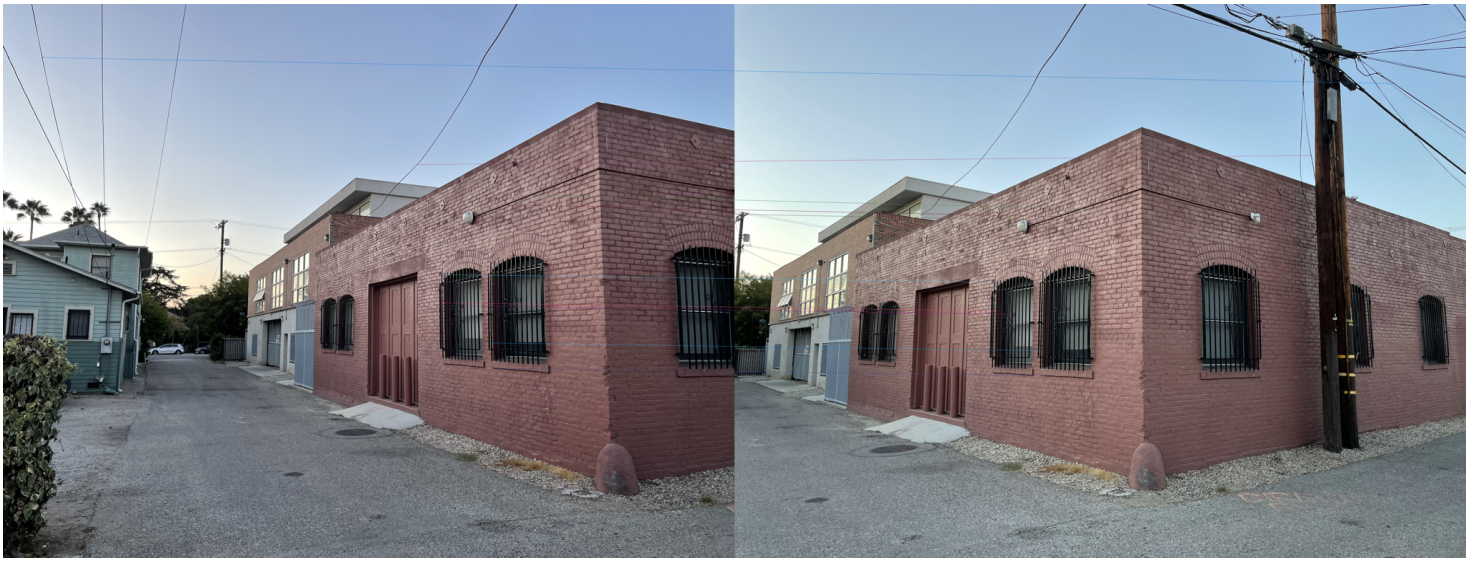
SIFT Features Image 2



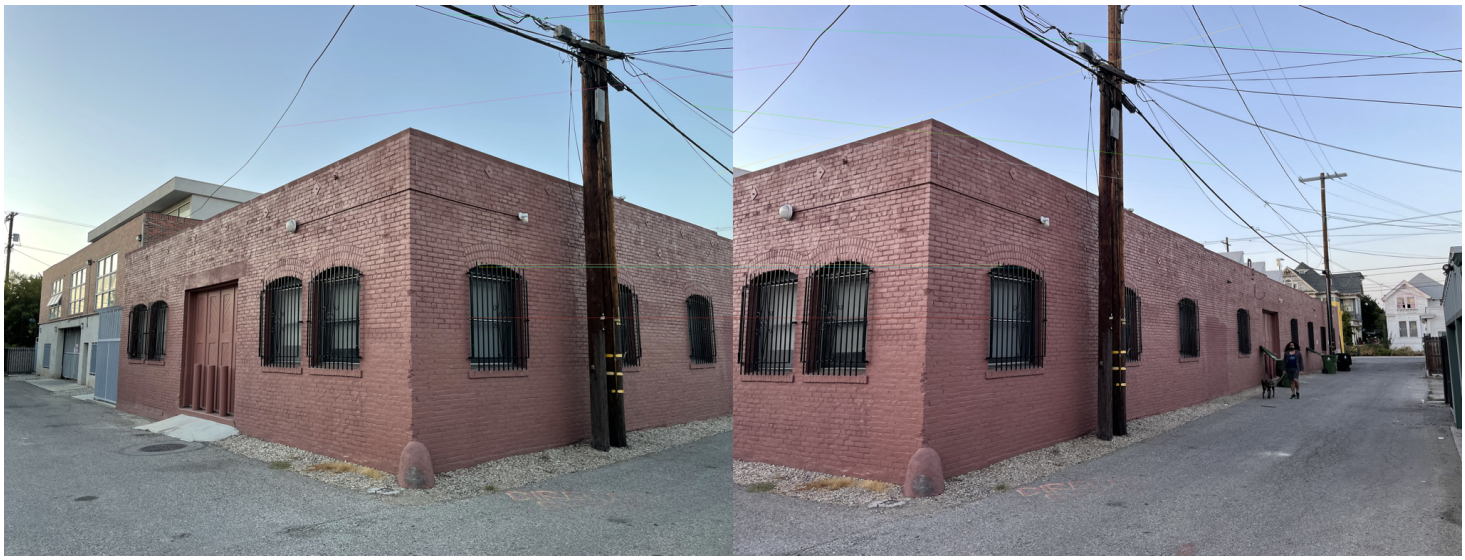
SIFT Features Image 3



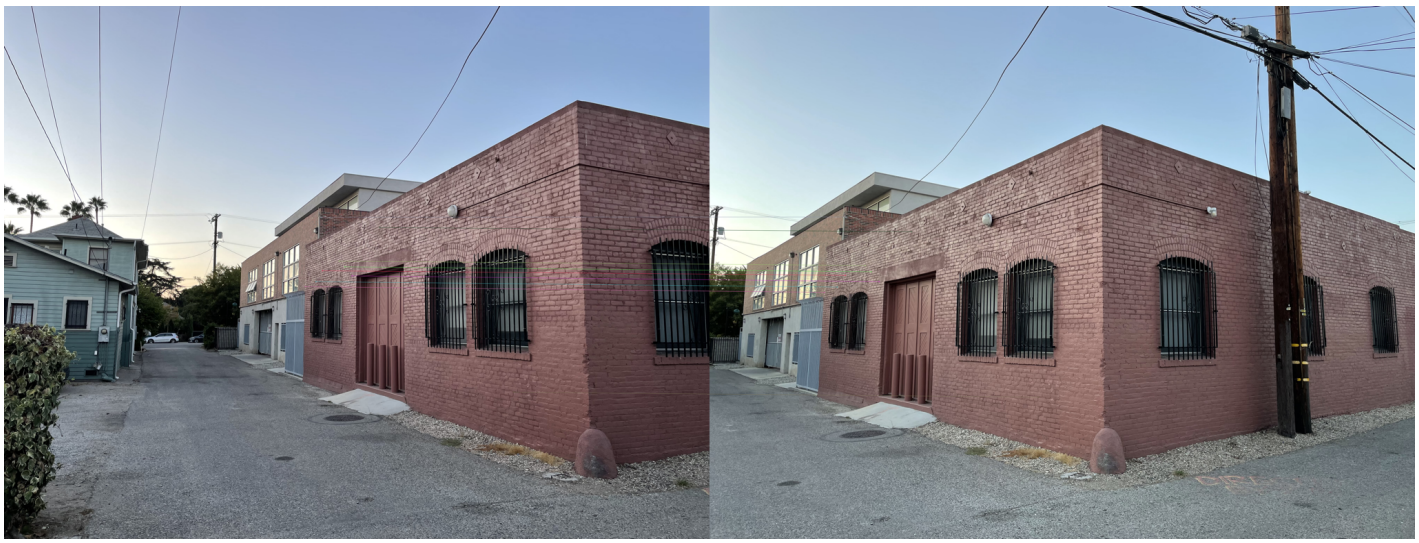
Top-10 Matches Image 1 & 2



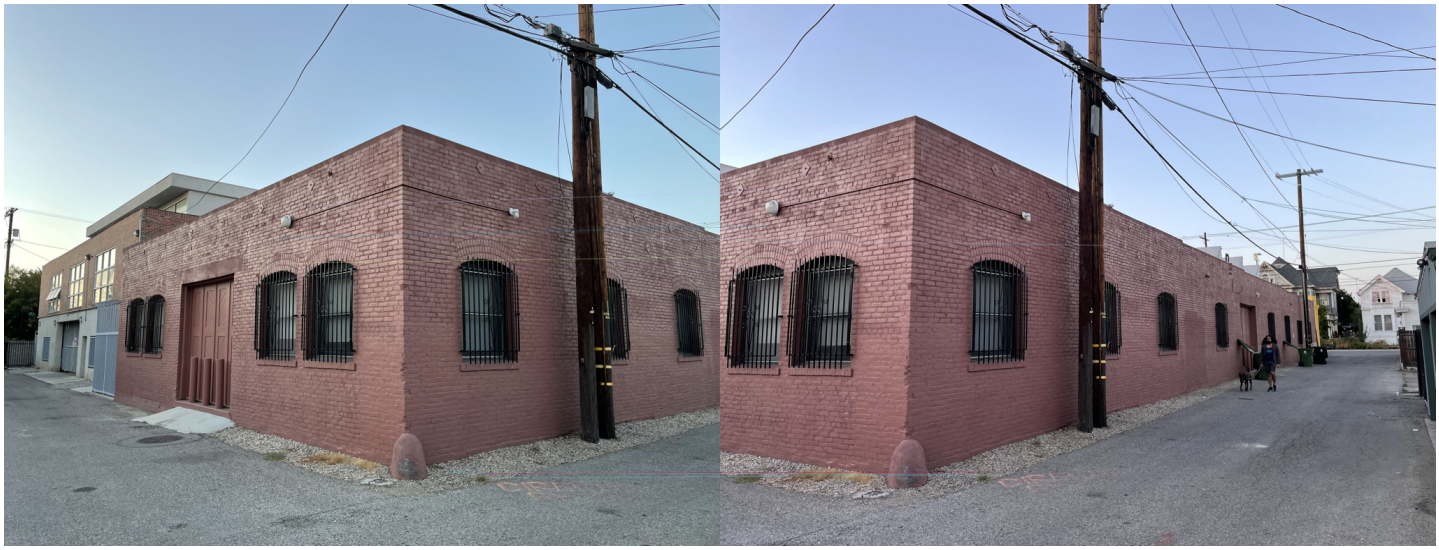
Top-10 Matches Image 2 & 3



Top-10 Inliers Image 1 & 2



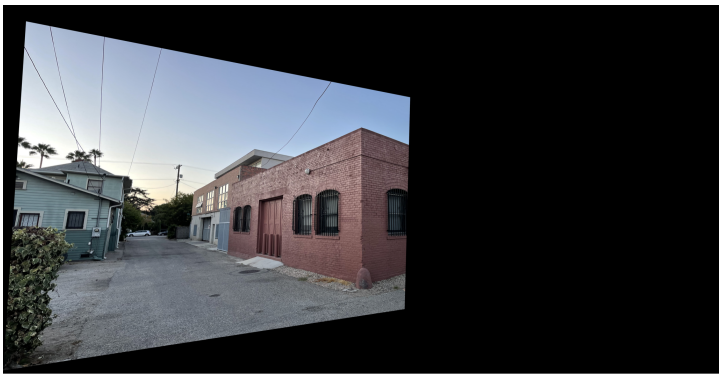
Top-10 Inliers Image 2 & 3



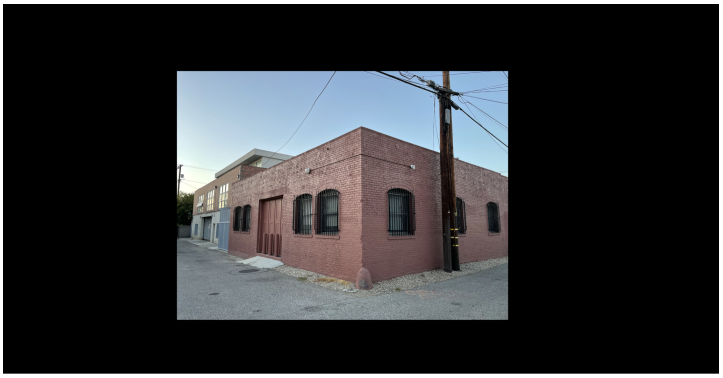
$$H_{0\to1} = \begin{bmatrix} 1.337056 & -0.103439 & -917.8802 \\ 0.211671 & 1.197784 & -475.3298 \\ 0.0000818 & 0.0000024 & 1.000000 \end{bmatrix}$$

$$H_{1\to2} = \begin{bmatrix} 1.530720 & -0.054279 & -1515.1843 \\ 0.243685 & 1.336173 & -686.5062 \\ 0.0001264 & 0.0000093 & 1.000000 \end{bmatrix}$$

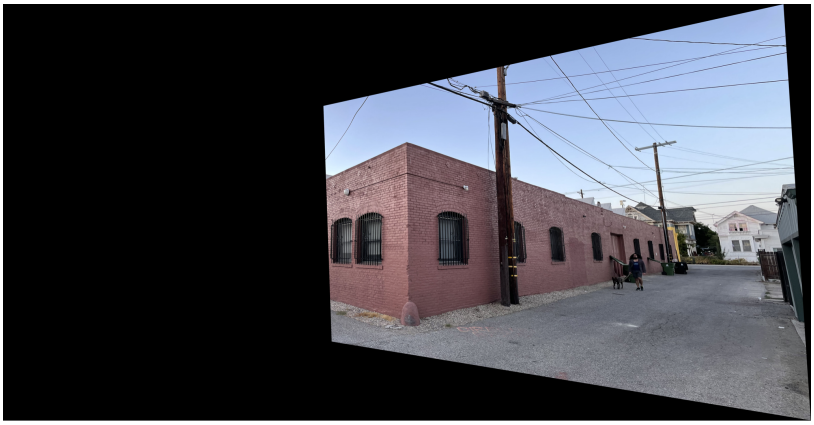
Warped Image 1



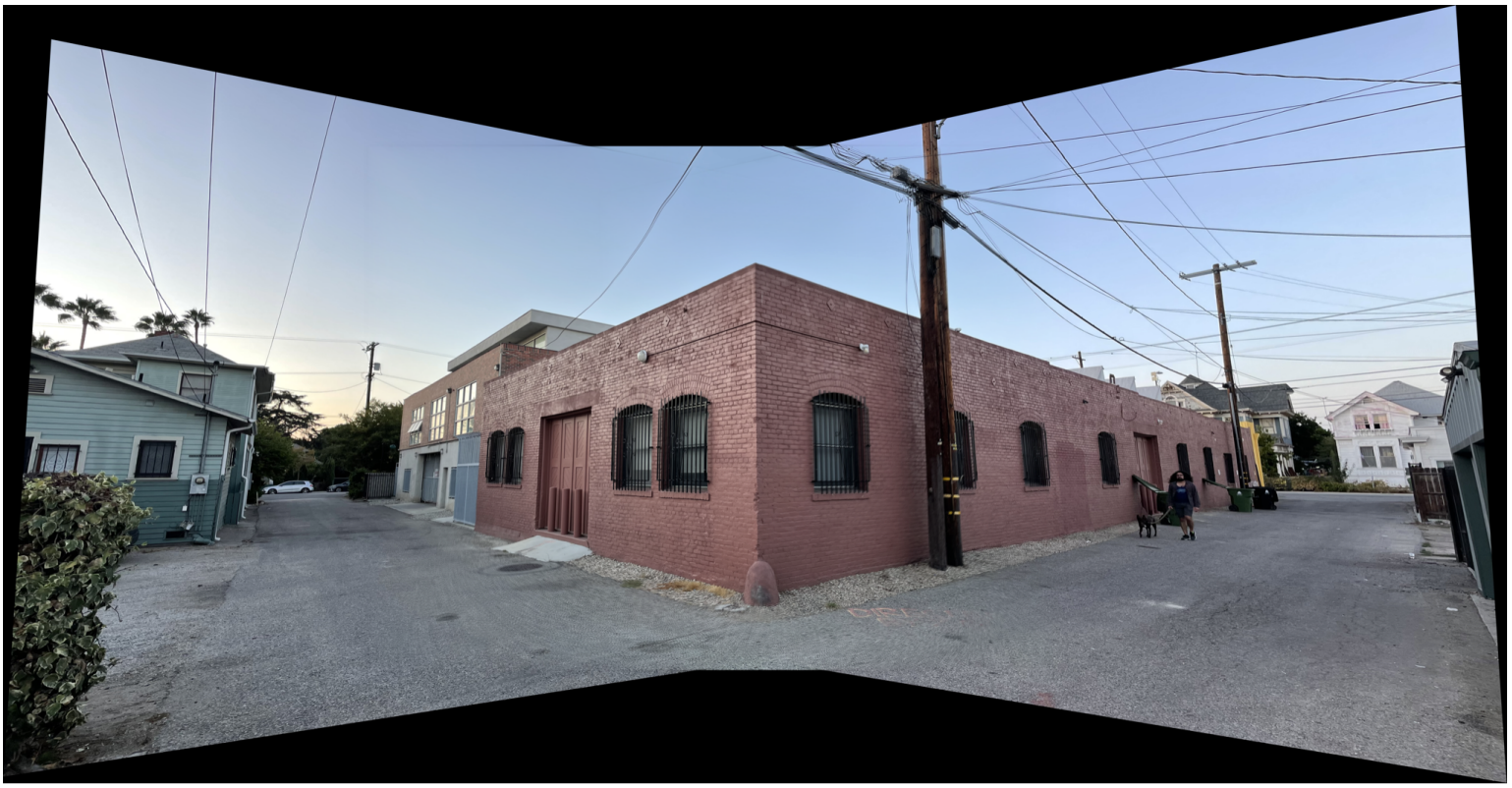
Warped Image 2



Warped Image 3



Final Panorama



```
"""
CSCI 677 - Advanced Computer Vision
Fall 2025 - Homework 2

Author: Xinjie Chen
Date: 29 September 2025
USC ID: 94216912368
Email: xinjie@usc.edu
"""

import os
import glob
import math
import cv2 as cv
import numpy as np

# Set input directory for images and output directory for all results.
DATA_DIR = "./data"
OUT_DIR  = "./out"
os.makedirs(OUT_DIR, exist_ok=True)

# Basic runtime settings: scale images, set ratio/RANSAC thresholds, and define image order.
RATIO = 0.70           # 0.70~0.80; smaller = stricter matches
RANSAC_THRESH = 3.0    # px; 3~6 reasonable
IMG_LIST = ["001.JPG", "002.JPG", "003.JPG"]

def read_images():

    # Read all color images from the data folder
    paths = []
    if IMG_LIST is not None:
        for item in IMG_LIST:
            paths.append(os.path.join(DATA_DIR, item))
    else:
        raise RuntimeError("Provide IMG_LIST")

    # Generate and return path list, color image list, and grayscale image list
    imgs_bgr, imgs_gray = [], []
    for p in paths:
        img = cv.imread(p, cv.IMREAD_COLOR)
        if img is None:
            raise FileNotFoundError(f"Cannot read image: {p}")
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        imgs_bgr.append(img)
        imgs_gray.append(gray)
```

```
    return paths, imgs_bgr, imgs_gray

def detect_sift(gray):

    # Detect keypoints and compute descriptors on grayscale images using SIFT
    sift = cv.SIFT_create()
    kpts, desc = sift.detectAndCompute(gray, None)
    return kpts, desc

def visualize_keypoints(img_bgr, kpts, out_path):

    # Draw SIFT keypoints on the color image and save it;
    # circle size shows scale, sector indicates orientation.
    vis = cv.drawKeypoints(
        img_bgr, kpts, None,
        flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
    )
    cv.imwrite(out_path, vis)

def ratio_test_knn(desc1, desc2, k=2, ratio=RATIO):

    # Match features between neighboring images using BFMatcher and Lowe's ratio test,
    # returning good matches (for RANSAC) and raw matches (for counting).
    bf = cv.BFMatcher(cv.NORM_L2, crossCheck=False)
    raw = bf.knnMatch(desc1, desc2, k=k)
    good = []
    for pair in raw:
        if len(pair) < 2:
            continue
        m, n = pair[0], pair[1]
        if m.distance < ratio * n.distance:
            good.append(m)
    return good, raw

def draw_topK_matches(img1, kpts1, img2, kpts2, matches, K, out_path):

    # Visualize the top-K pre-RANSAC matches sorted by distance.
    matches_sorted = sorted(matches, key=lambda m: m.distance)[:K]
    vis = cv.drawMatches(
        img1, kpts1, img2, kpts2, matches_sorted, None,
        flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )
    cv.imwrite(out_path, vis)

def homography_ransac(kpts1, kpts2, matches):
```

```
    # Estimate homography H from ratio-test matches using RANSAC,
    # returning H, inlier mask, and reprojection errors.
    if len(matches) < 4:
        return None, None, None
    src = np.float32([kpts1[m.queryIdx].pt for m in matches]).reshape(-1,1,2)
    dst = np.float32([kpts2[m.trainIdx].pt for m in matches]).reshape(-1,1,2)
    H, mask = cv.findHomography(src, dst, cv.RANSAC, RANSAC_THRESH)

    # Compute reprojection error for each inlier (||H*x - x'||),
    # used to visualize the Top-10 inliers with the smallest errors.
    errs = None
    if H is not None and mask is not None:
        inliers = mask.ravel().astype(bool)
        if np.any(inliers):
            src_i = src[inliers][:,0,:]
            dst_i = dst[inliers][:,0,:]
            src_h = cv.perspectiveTransform(src_i.reshape(-1,1,2), H)[:,0,:]
            dif   = src_h - dst_i
            errs  = np.linalg.norm(dif, axis=1)
    return H, mask, errs

def draw_topK_inlier_matches_by_error(img1, kpts1, img2, kpts2,
                                      matches, mask, errs, K, out_path):

    #Draw the Top-K inlier matches with the smallest reprojection errors (after RANSAC).
    #mask: (N,1) array of 0/1 flags indicating which matches are inliers
    #errs: error array corresponding only to inliers (length = number of inliers)
    if mask is None or errs is None:
        return

    # Find inlier indices in the original matches list
    inliers_idx = np.flatnonzero(mask.ravel().astype(bool))
    if inliers_idx.size == 0:
        return

    # The order of errs follows the inlier sequence:
    # sort errs to get the Top-K inliers, then map them back to indices in the original
matches.
    order   = np.argsort(errs)                    # Small errors first
    K_eff   = min(K, order.size)                    # Handle case when inliers < K
    sel_idx = inliers_idx[order[:K_eff]]               # Map back to original match indices

    # Extract these matches and visualize
    sel = [matches[i] for i in sel_idx]
    vis = cv.drawMatches(img1, kpts1, img2, kpts2, sel, None,
                         flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

```
    cv.imwrite(out_path, vis)


def compose_h_to_anchor(H_pair_list, anchor_idx):

    # Compose homographies from neighboring images into cumulative transforms
relative to the anchor image.
    n = len(H_pair_list) + 1
    H_to_anchor = [None]*n
    H_to_anchor[anchor_idx] = np.eye(3, dtype=np.float64)

    # Chain from left to right: i -> i+1 -> ... -> anchor
    for i in range(anchor_idx-1, -1, -1):
        H_to_anchor[i] = H_to_anchor[i+1] @ H_pair_list[i]

    # Chain from right to left using inverse matrices
    for i in range(anchor_idx+1, n):
        H_inv = np.linalg.inv(H_pair_list[i-1])
        H_to_anchor[i] = H_to_anchor[i-1] @ H_inv

    return H_to_anchor

def warped_corners(shape_hw, H):

    # Transform the four image corners with H to anchor coordinates,
    # used to compute the overall bounding box after warping.
    h, w = shape_hw
    corners = np.float32([[0,0],[w,0],[w,h],[0,h]]).reshape(-1,1,2)
    wc = cv.perspectiveTransform(corners, H)[:,0,:]
    return wc

def compute_canvas_bounds(imgs_bgr, H_to_anchor):

    # Compute overall canvas bounds from transformed image corners,
    # and build a translation matrix to shift all coordinates into the canvas.
    all_pts = []
    for img, H in zip(imgs_bgr, H_to_anchor):
        wc = warped_corners(img.shape[:2], H)
        all_pts.append(wc)
    all_pts = np.vstack(all_pts)
    min_x = np.floor(np.min(all_pts[:,0])).astype(int)
    min_y = np.floor(np.min(all_pts[:,1])).astype(int)
    max_x = np.ceil(np.max(all_pts[:,0])).astype(int)
    max_y = np.ceil(np.max(all_pts[:,1])).astype(int)
    width  = max_x - min_x
    height = max_y - min_y
```

```
    # Translation matrix: shift all coordinates into [0, W) × [0, H) to avoid cropping.
    T = np.array([[1,0,-min_x],[0,1,-min_y],[0,0,1]], dtype=np.float64)
    return width, height, T

def blend_average(images_warped, masks_warped):

    # Blend all warped images by averaging pixels, using masks as weights to handle
overlaps.
    H, W = images_warped[0].shape[:2]
    acc = np.zeros((H, W, 3), np.float32)
    wgt = np.zeros((H, W, 1), np.float32)
    for I, M in zip(images_warped, masks_warped):

        # Convert single-channel mask M to (H,W,1) float (0/1) for broadcasting with 3-
channel images.
        m = (M > 0).astype(np.float32)[..., None]
        acc += I.astype(np.float32) * m
        wgt += m
    wgt[wgt == 0] = 1.0
    pano = (acc / wgt).clip(0, 255).astype(np.uint8)
    return pano

def _make_feather_weight(mask_warped, blur_ksize=51):

    # Generate feathering weights using distance transform and Gaussian blur
    # (larger weights farther from boundaries, smoother transitions).
    m = (mask_warped > 0).astype(np.uint8)

    # Distance transform: value is 0 at boundaries/uncovered areas, larger farther away.
    dist = cv.distanceTransform(m, distanceType=cv.DIST_L2, maskSize=5)

    # Normalize to [0,1]
    if dist.max() > 0:
        dist = dist / (dist.max() + 1e-6)

    # Blur to avoid sharp seams
    if blur_ksize > 1:
        dist = cv.GaussianBlur(dist, (blur_ksize, blur_ksize), 0)
        dist = dist / (dist.max() + 1e-6)
    return dist[..., None].astype(np.float32)


def feather_blend(images_warped, masks_warped):

    # Feather blending: apply distance-based weights to each warped image
```

```
    # and compute weighted average for smoother seams and fewer ghosts.
    H, W = images_warped[0].shape[:2]
    acc = np.zeros((H, W, 3), np.float32)
    wgt = np.zeros((H, W, 1), np.float32)

    weights = []
    for M in masks_warped:
        weights.append(_make_feather_weight(M))
    wsum = np.sum(weights, axis=0)
    wsum[wsum == 0] = 1.0
    for I, w in zip(images_warped, weights):
        acc += I.astype(np.float32) * (w / wsum)
    pano = acc.clip(0, 255).astype(np.uint8)
    return pano


def main():

    # Load data and convert to grayscale
    paths, imgs_bgr, imgs_gray = read_images()

    # SIFT detection and visualization
    kp_list, desc_list = [], []
    for idx, (p, bgr, gray) in enumerate(zip(paths, imgs_bgr, imgs_gray)):
        kpts, desc = detect_sift(gray)
        kp_list.append(kpts)
        desc_list.append(desc)
        print(f"[SIFT] {os.path.basename(p)}: keypoints = {len(kpts)}")
        visualize_keypoints(bgr, kpts, os.path.join(OUT_DIR, f"sift_{idx+1:02d}.png"))

    # Adjacent pair matching + RANSAC
    H_pairs = []
    for i in range(len(imgs_bgr)-1):
        imgA, imgB = imgs_bgr[i], imgs_bgr[i+1]
        kA, kB = kp_list[i], kp_list[i+1]
        dA, dB = desc_list[i], desc_list[i+1]

        # Raw match count before Ratio Test vs. good match count after filtering (for pre-
RANSAC statistics).
        good, raw = ratio_test_knn(dA, dB, k=2, ratio=RATIO)
        print(f"[Match pre-RANSAC] {i}-{i+1}: raw={len(raw)}  good={len(good)}")

        # Visualization of Top-10 matches before RANSAC.
        draw_topK_matches(imgA, kA, imgB, kB, good, 10,
                          os.path.join(OUT_DIR, f"preRANSAC_top10_{i}{i+1}.png"))
```

```
        # RANSAC to estimate H_{i->i+1}: print number of inliers and the homography
matrix.
        H, mask, errs = homography_ransac(kA, kB, good)
        if H is None:
            raise RuntimeError(f"Homography failed for pair {i}-{i+1}")
        inliers = int(mask.sum()) if mask is not None else 0
        print(f"[RANSAC] {i}->{i+1}: inliers={inliers}/{len(good)}")
        print(f"[H_{i}->{i+1}]:\n{H}\n")
        np.savetxt(os.path.join(OUT_DIR, f"H_{i}_to_{i+1}.txt"), H, fmt="%.6f")

        # Post-RANSAC: visualize Top-10 inlier matches with the smallest reprojection
errors.
        draw_topK_inlier_matches_by_error(
            imgA, kA, imgB, kB, good, mask, errs, 10,
            os.path.join(OUT_DIR, f"postRANSAC_top10_{i}{i+1}.png")
        )

        H_pairs.append(H)

    # Select anchor image and build cumulative homographies to it.
    # For 3 images, len=3, //2=1 → the middle image is chosen as anchor.
    anchor = len(imgs_bgr)//2
    H_to_anchor = compose_h_to_anchor(H_pairs, anchor)

    # Compute unified canvas size and translation matrix
    W, Hc, T = compute_canvas_bounds(imgs_bgr, H_to_anchor)
    print(f"[Canvas] width={W}, height={Hc}")

    # Save cumulative homography of each image to the anchor
    for i, H_i in enumerate(H_to_anchor):
        np.savetxt(os.path.join(OUT_DIR, f"H_img{i}_to_anchor.txt"), H_i, fmt="%.6f")

    # Warp each image to the canvas and export
    warped_images, warped_masks = [], []
    for i, img in enumerate(imgs_bgr):

        # Left-multiply cumulative H with translation matrix T to ensure all images fit
within the canvas
        M = T @ H_to_anchor[i]
        warped = cv.warpPerspective(img, M, (W, Hc))

        # Generate mask under the same transform (value > 0 means the image covers that
canvas pixel)
        h, w = img.shape[:2]
        mask = np.ones((h, w), np.uint8)*255
        mask_warped = cv.warpPerspective(mask, M, (W, Hc))
```

```
        warped_images.append(warped)
        warped_masks.append(mask_warped)
        cv.imwrite(os.path.join(OUT_DIR, f"warped_{i}.png"), warped)

    # Blend and export the final panorama

    # pano = blend_average(warped_images, warped_masks)

    pano = feather_blend(warped_images, warped_masks)

    cv.imwrite(os.path.join(OUT_DIR, "panorama.png"), pano)
    print("[Done] Results saved to ./out")

if __name__ == "__main__":
    main()
```