Comprehensive Report on Image Stitching Pipeline

Abstract

This report provides a detailed analysis of a Python-based image stitching pipeline implemented in a Jupyter Notebook environment. The code leverages libraries such as OpenCV, NumPy, and Matplotlib to create panoramic images by automatically aligning and combining multiple overlapping images. The pipeline encompasses key computer vision techniques including feature detection (AKAZE), feature matching, robust homography estimation using RANSAC, and image warping. This document explains the underlying principles of image stitching, breaks down the architecture of the provided code, details the functionality of each Python function, and discusses the overall workflow and potential results. It serves as comprehensive documentation for understanding and potentially extending the implemented solution.

Contents

1 Introduction

Image stitching, or photo stitching/mosaicing, is a fundamental technique in computer vision and computational photography used to combine multiple images with overlapping fields of view into a single, wider panorama or higher-resolution image. This process is essential for applications ranging from creating immersive landscape photographs and virtual tours to medical imaging and satellite image composition.

The core challenge lies in accurately aligning the input images despite differences in perspective, lighting, and potential lens distortions. Modern image stitching pipelines typically automate this process through a series of steps:

1. **Feature Detection:** Identifying distinctive keypoints (corners, blobs, edges) within each image.
2. **Feature Description:** Computing a descriptor vector for each keypoint, capturing the appearance of its local neighborhood.
3. **Feature Matching:** Finding corresponding keypoints between overlapping images based on descriptor similarity.
4. **Transformation Estimation:** Calculating the geometric transformation (e.g., homography) that aligns one image with another, often using robust methods like RANSAC to handle outliers from incorrect matches.
5. **Image Warping:** Applying the estimated transformation to one or more images to bring them into a common coordinate system.
6. **Blending/Compositing:** Combining the aligned images smoothly to create the final seamless panorama, often involving techniques to minimize visual artifacts at the seams.

The provided Python code implements such a pipeline, primarily using the AKAZE feature detector/descriptor and homography-based alignment with RANSAC. This report dissects this implementation function by function.

2 Core Concepts in Image Stitching

Understanding the fundamental techniques employed is crucial for appreciating the code's functionality.

2.1 Feature Detection and Description

The process begins by finding salient points (keypoints) in each image that can be reliably detected even under viewpoint or lighting changes. Descriptors are then computed for these keypoints to represent their local appearance uniquely.

- **Keypoints:** Points of interest in an image (e.g., corners, edges, blobs) that are stable under transformations.
- **Descriptors:** Vectors that summarize the image patch around a keypoint, designed to be invariant to certain changes (like rotation, scale, illumination).
- **AKAZE:** The detector/descriptor used in this code (Accelerated-KAZE). It's known for good performance, scale/rotation invariance, and efficiency, operating in a non-linear scale space. AKAZE produces binary descriptors, which are memory-efficient and allow for fast matching using Hamming distance. The code includes a function (`expand_binarize`) to convert these packed binary descriptors into explicit binary vectors suitable for L2 distance calculation (as squared L2 distance on these is equivalent to Hamming distance).

2.2 Feature Matching

Once descriptors are computed for all keypoints in both images, the next step is to find pairs of keypoints (one from each image) that likely correspond to the same 3D point in the scene.

- **Distance Metric:** For binary descriptors like AKAZE (after expansion), the squared Euclidean (L2) distance is used, which is equivalent to the Hamming distance for the original packed descriptors. The `compute_distance` function calculates pairwise squared L2 distances.
- **Nearest Neighbor Matching:** For each descriptor in the first image, find the descriptor(s) in the second image with the smallest distance.
- **Ratio Test (Lowe's Ratio Test):** A common technique to improve matching quality. For a descriptor in image 1, find its two nearest neighbors (NN1 and NN2) in image 2. If the ratio of distances (distance(NN1) / distance(NN2)) is below a certain threshold (e.g., 0.75), the match (descriptor 1 to NN1) is considered reliable. This helps discard ambiguous matches where a keypoint has multiple close potential matches. This is implemented in the `find_matches` function.

2.3 Homography Estimation

For images of planar scenes or images taken from the same viewpoint (pure rotation), the geometric relationship between corresponding points can be described by a homography – a 3x3 matrix `H`. A point `(x, y)` in image 1 maps to `(x', y')` in image 2 such that:

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

where $\sim$ denotes equality up to a scale factor.

- **Direct Linear Transform (DLT):** A standard algorithm to compute `H` from point correspondences. Each correspondence pair provides two linear constraints on the elements of `H`. With at least 4 non-collinear point pairs, `H` can be estimated (up to scale) by solving a system of linear equations, typically formulated as $Ah = 0$. The solution is often found using Singular Value Decomposition (SVD) of the matrix A. The `fit_homography` function implements this.

2.4 RANSAC (RANdom SAmple Consensus)

Feature matching often produces incorrect matches (outliers). RANSAC is a robust estimation technique used to find the model parameters (here, the homography matrix `H`) that best fit the data in the presence of outliers.

- **Workflow:**
  1. Randomly select a minimal subset of matched points required to estimate the model (4 pairs for homography).
  2. Compute the homography `H` using this minimal set (`fit_homography`).
  3. Determine the set of inliers: Transform all points from image 1 using `H` and calculate the distance to their matched points in image 2. Points with distance below a threshold (`eps`) are inliers.
  4. Repeat steps 1-3 for a fixed number of iterations (`nIters`).
  5. Keep the homography `H` that yielded the largest number of inliers.
  6. (Optional but recommended) Re-estimate the homography using *all* identified inliers from the best model to get a more accurate result (`bestRefit`).
- The `RANSAC_fit_homography` function implements this iterative process.

2.5 Image Warping and Blending

Once the aligning homography `H` is found, one image needs to be transformed (warped) into the coordinate system of the other. Then, the images are combined.

- **Canvas Creation:** Determine the bounding box required to contain both the reference image and the warped image. Create a blank canvas of this size.
- **Perspective Warping:** Use the homography `H` (potentially combined with a translation matrix to fit onto the canvas) to warp the source image onto the canvas using functions like OpenCV's `cv2.warpPerspective`. This function often uses interpolation (like bilinear) to determine pixel values at non-integer coordinates.
- **Blending:** Combine the pixel values in the overlapping regions. The provided code uses a simple overlay (copying non-zero warped pixels onto the reference image placed on the canvas). More advanced techniques like alpha blending or multi-band blending can produce smoother transitions. The `warp_and_combine` function (from cell 9, which seems more aligned with standard stitching logic than the function in cell 18) handles canvas creation and warping.

2.6 Bilinear Interpolation

When warping an image, pixel coordinates often map to non-integer locations in the source image. Bilinear interpolation is used to estimate the pixel value at these fractional coordinates by taking a weighted average of the four nearest pixel neighbors. The `bilinear` function provides an implementation, although OpenCV's `cv2.warpPerspective` handles this internally.

3 Code Architecture and Workflow

The notebook follows a standard image stitching pipeline structure:

1. **Setup and Imports:** Load necessary libraries (`os`, `cv2`, `numpy`, `matplotlib`). Mount Google Drive if running in Colab.
2. **Utility Functions:** Define helper functions for common tasks:
   - Image I/O: `read_img`, `save_img`.
   - Display: `show_img`.
   - Geometric Transformation: `homography_transform`.
   - Keypoint/Match Handling: `kps_to_matrix`, `get_match_points`.
   - Descriptor Handling: `expand_binarize`, `get_AKAZE`.
   - Distance Calculation: `compute_distance`.
   - Interpolation: `bilinear`.
3. **Core Algorithm Functions:** Define functions central to the stitching process:
   - `fit_homography`: Estimates homography from 4+ points using DLT/SVD.
   - `find_matches`: Implements descriptor matching with the ratio test.
   - `RANSAC_fit_homography`: Robustly estimates homography using RANSAC.
   - `draw_matches`: Visualizes found correspondences.
   - `warp_and_combine` (cell 9) / `warp_and_stitch` (cell 18): Warps and combines images. (The logic in cell 9 seems more standard for stitching image 1 onto image 2's plane).

4. **Main Execution:**
    - Specify image paths and load the two images (`I1`, `I2`).
    - Extract AKAZE keypoints and descriptors for both images (`get_AKAZE`).
    - Compute distances between all descriptor pairs (`compute_distance`).
    - Find reliable matches using the ratio test (`find_matches`).
    - Visualize the matches (`draw_matches`, `show_img`).
    - Prepare matched point coordinates (`get_match_points`).
    - Estimate the robust homography using RANSAC (`RANSAC_fit_homography`).
    - Warp one image and combine it with the other using the estimated homography (`warp_and_combine` or `warp_and_stitch`).
    - Display the final stitched image (`show_img`).

4 Function Breakdown

4.1 Imports and Setup (Cell 1, Cell 10, Cell 15)

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive # Used in later cells
```

These lines import standard libraries for operating system interaction, computer vision (OpenCV), numerical operations (NumPy), plotting (Matplotlib), and Google Drive access (specific to Google Colab). Initial setup cells also mount Google Drive and define file paths.

4.2 Utility Functions (Cell 2)

4.2.1 `_bits` List Generation

This pre-computes binary representations for numbers 0-255, used later by `expand_binarize`.

4.2.2 `read_img(path)`

Reads an image from the specified `path` using OpenCV and ensures it's in RGB format (OpenCV reads as BGR by default, but the conversion here seems misplaced, it should be 'cv2.COLOR$_B$$GR2RGB$'$inside the function$

4.2.3 `save_img(img, path)`

Saves an image `img` to the specified `path` using OpenCV.

4.2.4 `show_img(image, *args, **kwargs)`

Displays an image using Matplotlib. It handles grayscale conversion (if needed) and converts BGR (OpenCV default) to RGB (Matplotlib expected format) for correct color display. It also turns off plot axes.

4.2.5 `homography_transform(X, H)`

Applies a 3x3 homography matrix `H` to a set of N 2D points `X` (shape Nx2). It converts points to homogeneous coordinates, performs matrix multiplication, and converts back to Cartesian coordinates by dividing by the third homogeneous component.

4.2.6 `get_match_points(kp1, kp2, matches)`

Takes keypoint matrices (`kp1`, `kp2`) and a list of match indices (`matches`, shape Kx2) and returns an array (shape Kx4) where each row contains the matched pair coordinates: `[x1, y1, x2, y2]`.

4.2.7 `kps_to_matrix(kps)`

Converts OpenCV's keypoint object list (`kps`) into a NumPy array (Nx4) containing point coordinates (`pt`), angle, and octave information for each keypoint.

4.2.8 `expand_binarize(desc)`

Converts packed binary descriptors (like AKAZE/ORB, shape NxF, dtype uint8) into an explicit binary matrix (shape Nx8F, dtype float) where each element is 0 or 1. This allows using L2 distance for matching, as squared L2 equals Hamming distance here.

4.2.9 `get_AKAZE(I)`

Detects AKAZE keypoints and computes their descriptors for an input image `I`. It returns the keypoints as a matrix (via `kps_to_matrix`) and the descriptors in expanded binary format (via `expand_binarize`).

4.2.10 `bilinear(image, row, col)`

Performs bilinear interpolation to find the intensity value at a fractional pixel coordinate (`row`, `col`) within a grayscale `image`.

4.3 Core Algorithm Functions

4.3.1 `fit_homography(XY)` (Cell 3)

Computes the 3x3 homography matrix `H` given at least 4 corresponding point pairs `XY` (shape Nx4).

- It sets up the linear system $Ah = 0$ based on the homography equation $x' \sim Hx$. Each point pair $(x, y) \leftrightarrow (x', y')$ provides two linear equations involving the 9 elements of H (represented as vector h).
- The matrix A (shape 2Nx9) is constructed.
- Singular Value Decomposition (SVD) of A is computed.
- The solution h is the right singular vector corresponding to the smallest singular value (the last column of V^T or last row of V).
- The resulting 9-element vector h is reshaped into the 3x3 matrix `H`.
- `H` is normalized by dividing by its bottom-right element $H[2, 2]$ to enforce the scale constraint $h_{33} = 1$.

4.3.2 `compute_distance(desc1, desc2)` (Cell 6)

Calculates the pairwise squared L2 distance between two sets of descriptor vectors `desc1` (NxF) and `desc2` (MxF).

- It uses the identity $\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^T b$.
- It computes the squared norm of each descriptor in `desc1` and `desc2`.

- It computes the dot product term $2 \times desc1 \cdot desc2^T$.
- It combines these terms using broadcasting to get the NxM distance matrix.

4.3.3 `find_matches(desc1, desc2, ratioThreshold)` (Cells 7 & 16)

Finds matches between descriptors `desc1` and `desc2` using the nearest neighbor distance ratio (NNDR) test.

- It first computes the distance matrix using `compute_distance`.
- For each descriptor `desc1[i]`, it finds the indices of the first (`nearest`) and second (`second_nearest`) nearest neighbors in `desc2` by sorting the i-th row of the distance matrix.
- It calculates the ratio: $distance(i, nearest)/distance(i, second\_nearest)$.
- If the `ratio` is less than `ratioThreshold`, the match `(i, nearest)` is considered valid and added to the `matches` list.
- Returns the list of good matches.

Note: This function is defined twice (Cell 7 and Cell 16) with identical logic.

4.3.4 `RANSAC_fit_homography(XY, eps, nIters)` (Cells 4 & 17)

Performs RANSAC to find a robust homography estimate.

- **Initialization:** Sets `bestH` to None (or identity) and `best_inliers` count to 0.
- **Iteration Loop (`nIters` times):**
  - Randomly sample 4 point correspondences from `XY`.
  - Compute a candidate homography `H` using `fit_homography` on the sample.
  - Calculate reprojection errors: Transform all points `XY[:,:2]` using `H` (via `homography_transform`) and compute the Euclidean distance to the corresponding points `XY[:,2:]`.
  - Count inliers: Find the number of points where the distance is less than the threshold `eps`.
  - Update best model: If the current number of inliers is greater than `best_inliers`, update `best_inliers` and store the current `H` as `bestH`.
- **Refinement:** After iterations, identify all inlier points corresponding to the `bestH`. Re-estimate the final homography using `fit_homography` on *all* these inliers.
- Returns the refined homography `bestH`.

Note: This function is defined twice. Cell 4 has placeholder 'pass' statements for helper functions. Cell 17 provides the actual implementation logic used in the main script, calling previously defined functions.

4.3.5 `draw_matches(img1, img2, kp1, kp2, matches)` (Cell 8)

Visualizes the matches found between two images.

- Stacks `img1` and `img2` vertically into a single output image.
- For each match `(i, j)` in the `matches` list:
  - Retrieves the coordinates of the i-th keypoint in `kp1` and the j-th keypoint in `kp2`.

- Adjusts the y-coordinate of the keypoint from `img2` by adding the height of `img1`.
- Draws a line connecting the pair of matched keypoints on the output image using a random color.

- Returns the image with matches drawn.

4.3.6 `warp_and_combine(img1, img2, H)` (Cell 9)

(Logic described here, seems more standard than Cell 18's function for mapping img1 to img2 space). Warps `img1` to align with `img2` using homography `H` and combines them.

- Calculates the coordinates of the corners of `img1` after being transformed by `H`.
- Determines the minimum and maximum x and y coordinates needed for the output canvas to contain both the original `img2` and the warped `img1`.
- Calculates the canvas dimensions (`new_width`, `new_height`).
- Defines a translation matrix (`translation_matrix`) to shift the origin so that all coordinates are positive within the canvas.
- Warps `img1` onto the canvas using the combined transformation (translation applied *after* the homography: `translation_matrix @ H`) via `cv2.warpPerspective`.
- Creates the blank canvas `V`.
- Places `img2` onto the canvas at its translated position.
- Overlays the non-black pixels of the `warped_img1` onto the canvas, effectively blending by replacement (where warped image 1 is "on top").
- Returns the stitched image `V`.

4.3.7 `warp_and_stitch(img1, img2, H)` (Cell 18)

(As implemented in the final cell). This function seems intended to perform stitching but its implementation logic appears to warp `img2` onto `img1`'s coordinate space based on the use of 'H' with 'img2' in 'cv2.warpPerspective'. However, the final combination places 'img2' directly onto the canvas and then overlays 'warped$_i mg1$'.$This specific implementation needs review for correctness relative to the estimated homogr$

Get image dimensions (`h1, w1, h2, w2`).

Define corners for `img2` and transform them using `H`. This suggests `H` maps from `img2` to `img1`.

Calculate canvas boundaries based on `img1`'s corners and the `warped_corners` of `img2`.

Define a translation matrix based on `min_x`, `min_y`.

Warp `img1` using only translation (essentially placing it on the canvas).

Warp `img2` using the combined transformation (`translation @ H`).

Combine by placing `img1_canvas` first, then overlaying non-zero pixels from `img2_canvas`.

Note: There's inconsistency between this function and 'warp$_a nd_c ombine$'$incell 9, and also within cell 18 itself f$
$this * function. Assuming 'best H' maps 'I1' to 'I2' space (standard convention), this function should ideally yu$

4.4 Main Execution Script (Cells 10, 15, 16, 17, 18)

These cells orchestrate the pipeline:

- **Cell 10/15:** Mounts Google Drive, sets the root path, defines image names, and loads `I1` and `I2`. Cell 15 additionally extracts AKAZE features and visualizes keypoints on the individual images.
- **Cell 16:** Finds matches between `desc1` and `desc2` using the defined `find_matches` function with a ratio threshold of 0.75.
- **Cell 17:** Gets the coordinates of the matched keypoints using `get_match_points`. It then calls `RANSAC_fit_homography` (using the logic defined within this cell) to compute the robust homography `bestH`. It also includes the definition for `warp_and_stitch` (which seems inconsistent as noted above). *Correction:* The call to 'RANSAC$_{fit_homography}$' uses the implementation logic defined within cell 17, not
$*Cell 18 : **Calls the$ `warp_and_stitch` $function (defined in cell 17) with$ `I1, I2,` and `bestH` $to create the$ `stitch`

5 Results and Discussion

The execution of the notebook produces several visual outputs:

- **Keypoint Visualization (Cell 15):** Displays the input images (`I1`, `I2`) with detected AKAZE keypoints overlaid as markers. This helps visualize the feature detection stage.
- **Match Visualization (Cell 17):** Shows the two input images stacked vertically, with lines connecting the keypoint pairs identified as matches after the ratio test. This allows visual inspection of the matching quality before RANSAC.
- **Stitched Image (Cell 18):** Displays the final panoramic image created by warping and combining `I1` and `I2` using the RANSAC-estimated homography.

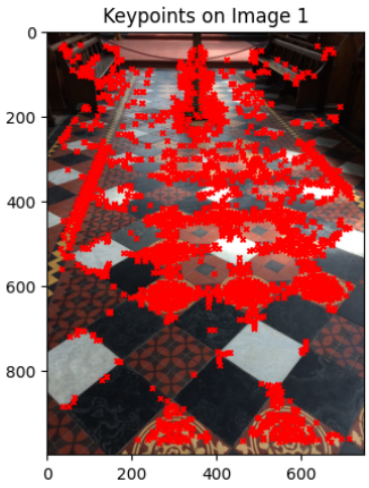
Discussion Points:

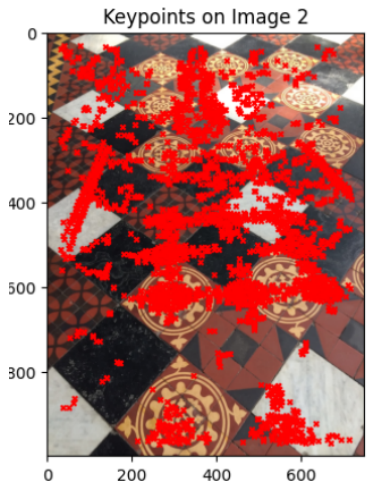
- **Blending:** The code uses simple replacement for blending. This can lead to visible seams or ghosting artifacts in the overlap region, especially if there are exposure differences or minor misalignments. More sophisticated blending methods (e.g., linear blending, multi-band blending, Poisson blending) would improve the visual quality.
- **Homography Limitation:** The homography model assumes a planar scene or pure camera rotation. If there is significant parallax (due to camera translation and non-planar scene elements), a simple homography will not perfectly align the images, leading to misalignments (ghosting).
- **Feature Detector Choice:** AKAZE is used here. Other detectors/descriptors (SIFT, SURF, ORB) might perform differently depending on the image content and transformations involved.
- **RANSAC Parameters:** The RANSAC threshold (`eps`) and number of iterations (`nIters`) influence the robustness and accuracy of the homography estimation. These might need tuning for different image pairs.

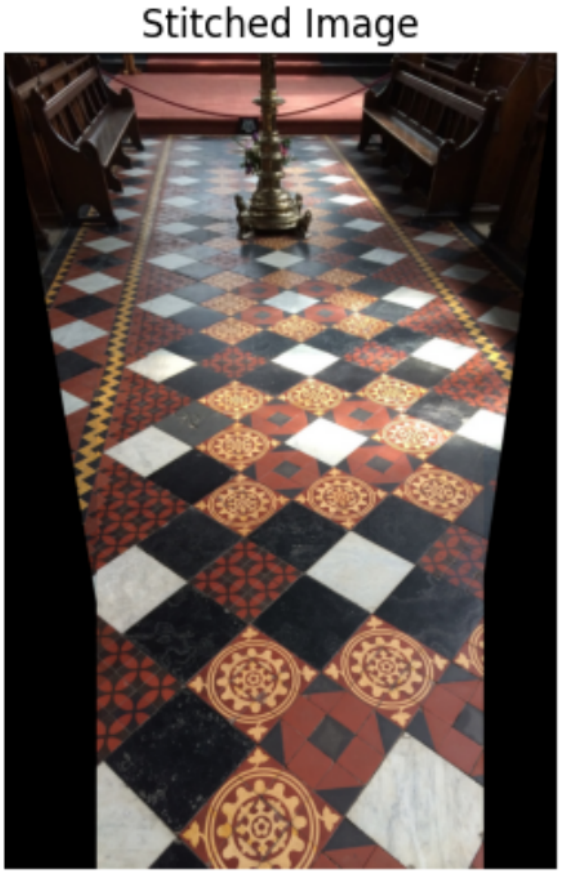
- **Warping Function Consistency:** As noted, there appears to be an inconsistency or potential error in the logic of the final `warp_and_stitch` function implementation compared to standard practices and the earlier `warp_and_combine` definition. The direction of the homography mapping (`I1 -> I2` or `I2 -> I1`) and which image is warped needs to be consistent for correct results. Based on the standard RANSAC setup using `XY[:,:2]` (points from image 1) and `XY[:,2:]` (points from image 2), `bestH` typically maps image 1 points to image 2 coordinates. Therefore, `warp_and_combine` (cell 9) which warps `img1` using `H` seems more appropriate.

6 Conclusion

The provided Python code successfully implements a basic image stitching pipeline using feature-based alignment. It demonstrates the core steps involving AKAZE feature detection/description, descriptor matching with a ratio test, robust homography estimation via RANSAC, and perspective warping for combining images. While functional, the visual quality of the stitched result could be enhanced by incorporating more advanced blending techniques. Furthermore, the accuracy for non-planar scenes or scenes with significant camera translation could be improved by using more complex models than a single homography (e.g., bundle adjustment, spatially varying warps). The discrepancy in the warping function implementations also warrants attention for ensuring correctness. Overall, the code serves as a solid foundation for understanding and implementing automatic image stitching.


Keypoints on Image 1
0
200
400
600
800
0
200
400
600


Keypoints on Image 2
0
200
400
500
300
0
200
400
600


Stitched Image