This Flask API is designed to process microscope images by performing image stitching, region of interest (ROI) selection, zooming, and auto-focus simulation using OpenCV.

**Approach Taken**

**1. Flask Setup & Image Upload**

* Created a Flask app with multiple endpoints for processing images.
* Used UPLOAD\_FOLDER to store uploaded images and processed results.
* Implemented save\_uploaded\_images() to handle multiple file uploads.

**2. Image Stitching Using Feature Matching & Homography**

* Implemented **ORB (Oriented FAST and Rotated BRIEF)** for feature detection and matching.
* Used **Homography transformation** to align overlapping images.
* Applied OpenCV’s cv2.warpPerspective() to transform images and merge them.

**3. Region of Interest (ROI) Selection & Zooming**

* Allowed users to select an ROI by specifying coordinates (x, y, width, height).
* Applied OpenCV’s cv2.resize() for zooming in on the selected region.

**4. Auto-Focus Simulation**

* Applied a **sharpening kernel** using OpenCV’s cv2.filter2D() to enhance image clarity.

**Challenges Faced**

**1. Feature Matching Issues**

* Initially, some images failed to align due to poor feature matches.
* Solution: Used **ORB** as a default feature detector, but allowed flexibility to switch to **SIFT/SURF**.

**2. Homography Computation Failing**

* Some images had **insufficient matches**, leading to failed homography calculations.
* Solution: Implemented error handling for cases where matches were too low.

**3. Zoomed Images Losing Quality**

* Resizing with poor interpolation resulted in pixelation.
* Solution: Used cv2.INTER\_CUBIC for better zoom quality.

**4. Slow Image Processing**

* Stitching multiple images took longer than expected.
* Solution: Optimized by **sorting images based on feature matches** for better alignment order.

**Optimizations Applied**

**Modular Code Structure**

* Organized code into reusable functions for **image saving, feature matching, and stitching**.
* This made the code **cleaner and easier to debug**.

**Efficient Feature Matching**

* Used **ORB** for speed while providing **SIFT/SURF** as alternatives for better accuracy.

**Homography Filtering**

* Ensured only **good matches** were used for homography calculation to improve alignment.

**Error Handling & Logging**

* Added checks for file uploads, feature detection failures, and homography errors.
* This prevented crashes and provided meaningful error messages.

**Conclusion**

This project helped in understanding **Flask API development, OpenCV image processing, and feature matching techniques**. By optimizing feature detection, error handling, and processing speed, **API is build** for microscope image analysis.

**NOTE:**

Here we used some sample images for development. This API is built in such a way that you can upload any image of your choice where there is an upload option you get by running

<http://127.0.0.1:5000/upload_images>

this and the output of stitched image can be seen by running this.

<http://127.0.0.1:5000/stitch_images>