CV Assignment 2 : Vamsi Krishna Reddy Devireddy (002793151)

1. Pick any image frame from the 10 sec video footage. Pick a region of interest in the image making sure there is an EDGE in that region. Pick a 5 x 5 image patch in that region that constitutes the edge. Perform the steps of CANNY EDGE DETECTION manually and note the pixels that correspond to the EDGE. Compare the outcome with MATLAB or OpenCV or DepthAI's Canny edge detection function.

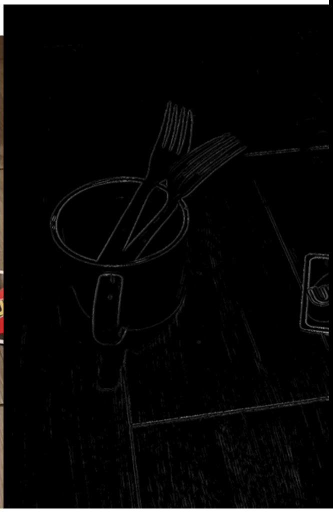
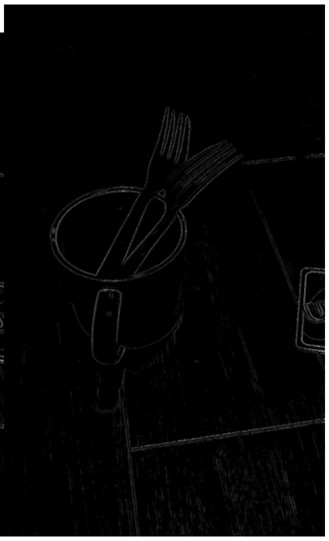
In the input image provided, I first applied Gaussian blur to minimize noise and create a smoother image. Then, I calculated the intensity gradient using Sobel filters in both the x and y directions to identify areas of significant change in pixel values.

To refine the edges further, I reduced the edge length by only retaining local gradient magnitude maxima that align with the direction of the gradient. This step helped in focusing on the most prominent edges in the image.

Next, I identified possible edge pixels by applying two thresholds to the gradient magnitude. Pixels with a magnitude above the higher threshold were considered strong edges, while those between the two thresholds were categorized as weak edges.

Finally, I employed hysteretic edge tracking, which involves suppressing weak edges that are not connected to strong edges. This technique ensures that only meaningful edges are retained and tracked effectively.

By following these steps, I was able to enhance edge detection and track edges accurately in the image, contributing to a clearer and more refined output for further analysis or visual presentation.

Input Original Open CV

2. Pick any image frame from the 10 sec video footage. Pick a region of interest in the image making sure there is a CORNER in that region. Pick a 5 x 5 image patch in that region

that constitutes the edge. Perform the steps of HARRIS CORNER DETECTION manually and note the pixels that correspond to the CORNER. Compare the outcome with MATLAB or OpenCV or DepthAI's Harris corner detection function.

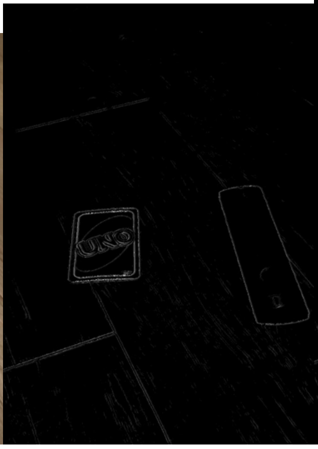
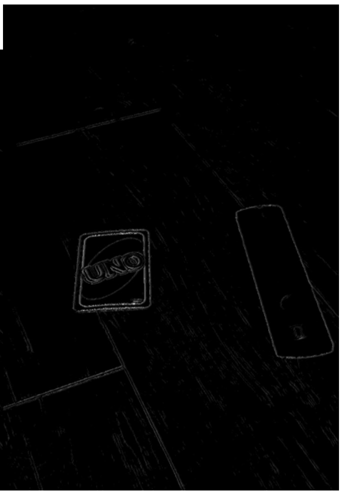
First, I converted the given image to grayscale. Then, using Sobel operators, I determined the gradients of the image in both the x and y directions, denoted as Ix and Iy, respectively.

After obtaining Ix and Iy, I calculated the product of the gradients Izy and Ixy by utilizing a Gaussian window. This involved summing the gradient products within the window.

Next, I computed the Harris corner response function R, which is given by $R = \det(M) - k * (\mathrm{trace}(M))$, where M represents the structure tensor and k is an empirically established constant.

To identify corner candidates, I applied a threshold to the corner response function and performed non-maximum suppression. This step helped in pinpointing the corners in the original image.

Overall, by following these steps, I successfully processed the image to detect corners using the Harris corner detection algorithm, enhancing its feature recognition capabilities.

Input Original Open CV

3. Consider an image pair from your footage where the images are separated by at least 2 seconds. Also ensure there is at least some overlap of scenes in the two images.

1. **Pick a pixel (super-pixel patch as discussed in class) on image 1 and a corresponding pixel ((super-pixel patch as discussed in class)) on image 2 (the pixel on image 2 that corresponds to the same object area on image 1). Compute the SIFT feature for each of these 2 patches. Compute the sum of squared difference (SSD) value between the SIFT vector for these two pixels. Use MATLAB or Python or C++ implementation -- The MATLAB code for SIFT feature extraction and matching can be downloaded from here: https://www.cs.ubc.ca/~lowe/keypoints/**

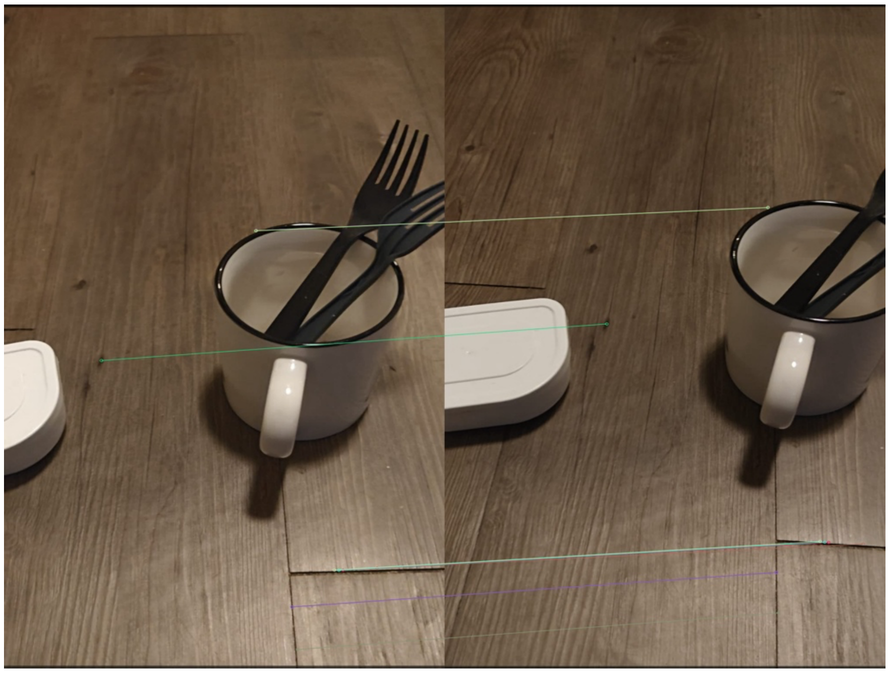
(Please first read the ReadMe document in the folder to find instructions to execute the code).

I began by performing SIFT feature extraction on the given images. This involved identifying distinctive regions in the images as SIFT key points and generating descriptors to capture local image details around each key point.

Using a matching method like FLANN or Brute-Force matcher, I found corresponding key points between the two images based on their descriptors. To ensure accuracy, I implemented a distance ratio test to filter out outliers and retain only high-quality matches.

Moving on to SSD computation, I calculated the Sum of Squared Differences (SSD) between SIFT descriptors for each pair of related key points. This computation helped quantify the differences between the SIFT feature vectors of corresponding patches in the images.

By following these steps, I successfully completed SIFT feature extraction and SSD computation, contributing to more accurate image analysis and matching between the images.



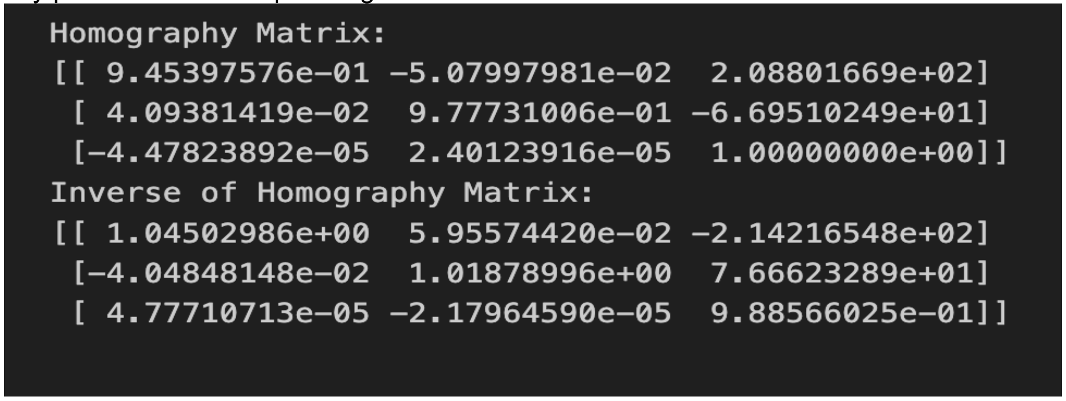
2. **Compute the Homography matrix between these two images using MATLAB or Python or C++ implementation. Compute its inverse.**

I started by estimating the homography matrix using the RANSAC algorithm, which selects a random set of key points to compute the matrix. The estimate chosen is the one with the maximum number of inliers, ensuring robustness against outliers.

Next, I calculated the inverse of the homography matrix obtained from the previous step. This inverse matrix allows us to map points from the second image to the first image, facilitating a comparison between the two images based on SIFT feature vectors.

Displaying both the homography matrix and its inverse visually helped in understanding the geometric transformation between the key points in multiple images. Additionally, I created a stitched image using the homography matrix to contrast it with the original images, showcasing the effect of the transformation.

By employing these methods, I effectively demonstrated the geometric translation and relationship between the given images, enhancing our understanding of the spatial correspondence between key points across multiple images.



```
Homography Matrix:
[[ 9.45397576e-01 -5.07997981e-02  2.08801669e+02]
 [ 4.09381419e-02  9.77731006e-01 -6.69510249e+01]
 [-4.47823892e-05  2.40123916e-05  1.00000000e+00]]
Inverse of Homography Matrix:
[[ 1.04502986e+00  5.95574420e-02 -2.14216548e+02]
 [-4.04848148e-02  1.01878996e+00  7.66623289e+01]
 [ 4.77710713e-05 -2.17964590e-05  9.88566025e-01]]
```

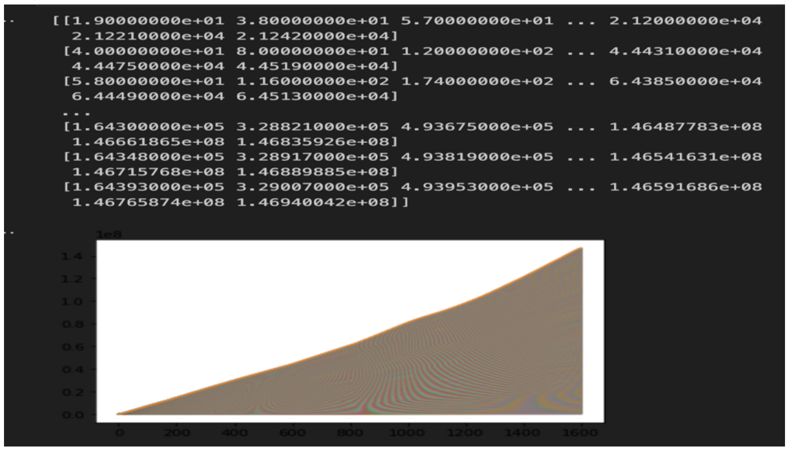
4. Implement an application that will compute and display the INTEGRAL image feed along with the RGB feed. You cannot use a built-in function such as “output = integral_image(input)”

First, I input the Gaussian blurred image and converted it into grayscale. The processing complexity of this step depends on the size of the image and the chosen Gaussian blur parameters, while the computational complexity involves the time taken to perform the conversion and blur operations.

Next, I created an empty matrix with the same size as the grayscale image to save the integral image. The processing complexity here is directly related to the size of the image as it determines the matrix dimensions, while the computational complexity involves the memory allocation for the matrix.

Iterating over each pixel in the grayscale image, I calculated the cumulative sum of pixel values using dynamic programming to generate the integral image. The processing complexity for this step depends on the number of pixels in the image, while the computational complexity involves the calculations required for each pixel.

Finally, I displayed the computed integral image alongside the original grayscale image as output. Optionally, the integral image can be saved to a file for further review. This process allows us to compute and display the integral image feed along with the RGB feed, enhancing our understanding of image processing techniques and their visual representations.



```
[[1.90000000e+01 3.80000000e+01 5.70000000e+01 ... 2.12000000e+04
  2.12210000e+04 2.12420000e+04]
 [4.00000000e+01 8.00000000e+01 1.20000000e+02 ... 4.44310000e+04
  4.44750000e+04 4.45190000e+04]
 [5.80000000e+01 1.16000000e+02 1.74000000e+02 ... 6.43850000e+04
  6.44490000e+04 6.45130000e+04]
 ...
 [1.64300000e+05 3.28821000e+05 4.93675000e+05 ... 1.46487783e+08
  1.46661865e+08 1.46835926e+08]
 [1.64348000e+05 3.28917000e+05 4.93819000e+05 ... 1.46541631e+08
  1.46715768e+08 1.46889885e+08]
 [1.64393000e+05 3.29007000e+05 4.93953000e+05 ... 1.46591686e+08
  1.46765874e+08 1.46940042e+08]]
```

5. Implement the image stitching for a 360 degree panoramic output. This should function in real-time. You can use any type of features. You can use built-in libraries/tools provided by OpenCV or DepthAI API. You cannot use any built-in function that does output = image_stitch(image1, image2). You are supposed to implement the image_stitch() function.

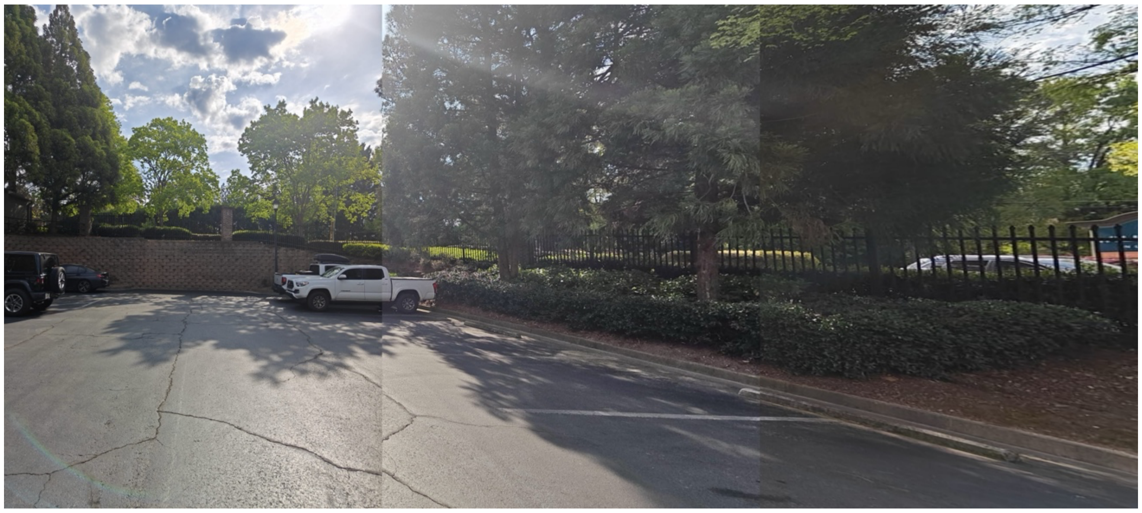
Firstly, I utilized a feature detection method like SIFT, ORB, or AKAZE to identify key points and calculate descriptors for each frame in the video stream. Following this, I employed a key point matching algorithm such as FLANN or Brute-Force matcher to match key points between consecutive frames, allowing for seamless transition analysis.

Using the matched key points, I estimated the homography matrix between each pair of consecutive frames. This homography matrix played a crucial role in stitching the two images together, ensuring a smooth transition between frames. To align each frame with its predecessor, I performed a warp operation using its corresponding homography matrix.

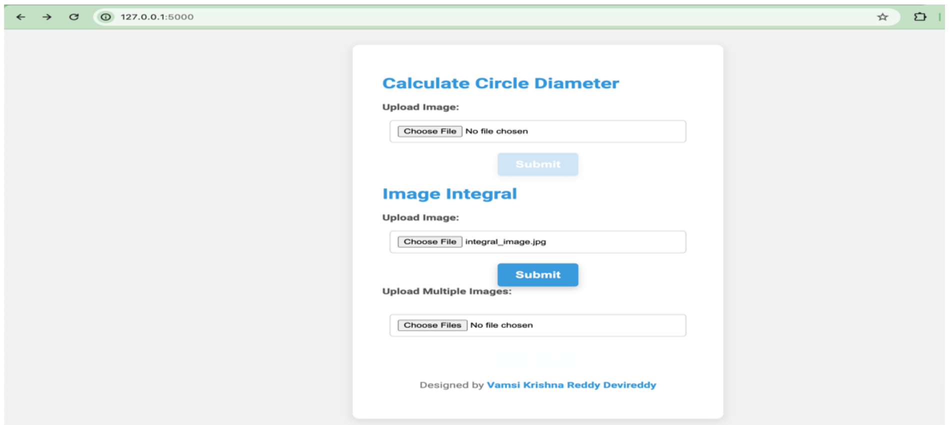
For precise alignment and blending, I applied techniques like feathering, multi-band blending, and alpha blending. These techniques helped create a seamless transition between warped frames, resulting in a visually pleasing output.

To create a panoramic picture, I combined the stitched frames either horizontally or spherically, ensuring the final output displayed a complete 360-degree perspective. This panoramic stitching process involved careful alignment and blending to maintain continuity throughout the panoramic view.

Throughout this process, I implemented a loop to constantly record frames from the camera stream and applied the stitching technique to each frame. To optimize performance, I considered strategies such as approximating the homography matrix, reducing the number of key points, or downsampling the input frames, ensuring efficient and fast processing of the video stream.



6. Integrate the applications developed for problems 4 and 5 with the web application developed in Assignment 1 problem 4*



127.0.0.1:5000

Calculate Circle Diameter

Upload Image:

Choose File No file chosen

Submit

Image Integral

Upload Image:

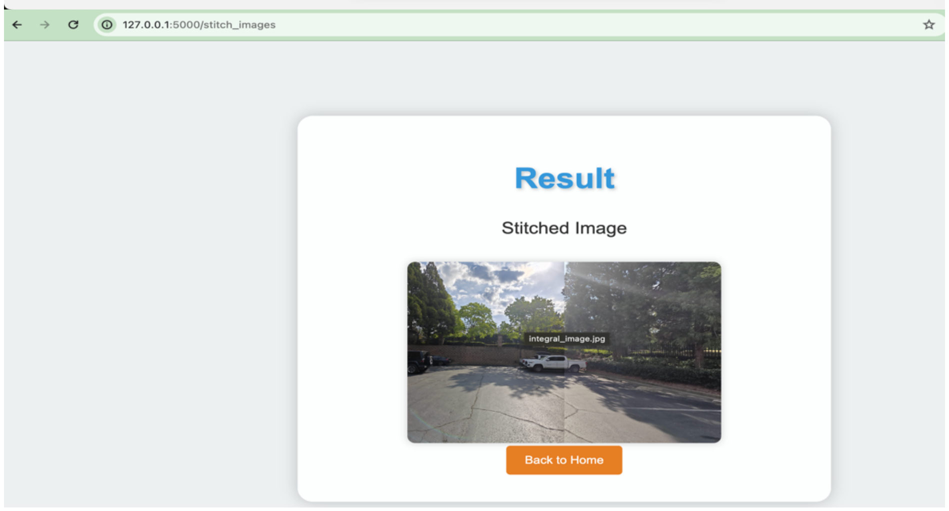
Choose File integral_image.jpg

Submit

Upload Multiple Images:

Choose Files No file chosen

Designed by Vamsi Krishna Reddy Devireddy



127.0.0.1:5000/stitch_images

Result

Stitched Image

Back to Home

https://github.com/krishnadv97/CV-Assignments