Assignment-2
Advanced Computer Vision

-By Ruchir Namjoshi
002803974

A1. CANNY EDGE DETECTION

To perform Canny edge detection algorithm manually on a 5x5 patch of an image. I used the following process

1. **Convert the image to grayscale:**

The input image is converted to a grayscale image. This is because edge detection typically operates on single-channel images to simplify the calculations.

2. **Define the Region of Interest (ROI) and extract the 5x5 patch:**

The function takes in a parameter "roi" which is a tuple defining the region of interest in the image with coordinates (x, y) and size `(w, h)`. The `patch_center` is a point within this region around which a 5x5 patch is extracted from the grayscale image. This patch is the region on which edge detection will be applied.

3. **Apply Gaussian blur to the patch:**

The extracted patch is then blurred using a Gaussian filter. This step helps reduce noise and spurious gradients in the image which could potentially be detected as edges. The kernel size for the Gaussian blur is 5x5, and the standard deviation of the Gaussian kernel is `1.4`.

4. **Compute gradients using Sobel operators:**

The gradients of the blurred patch are computed in both the x and y directions using the Sobel operator. The Sobel operator is a discrete differentiation operator that computes an approximation of the gradient of the image intensity function. The `np.hypot` function is then used to calculate the magnitude of the gradient vector at each pixel, combining both directional gradients.

5. **Normalize the magnitude:**

The gradient magnitudes are normalized to the range of 0 to 255. This is to ensure that the edge magnitudes can be compared to a threshold value in a meaningful way.

6. **Apply a simple threshold to detect edges:**

A thresholding step is applied to determine where edges are present in the patch. Any pixels with a gradient magnitude greater than the threshold `threshold_value` are marked as edges (pixel value set to 255), and the rest are set to 0 (no edge)

.

7. Compare with OpenCV's Canny:

Finally, the function compares the manually detected edges with those detected by OpenCV's built-in Canny edge detector on the same patch. This serves as a way to verify the effectiveness of the manual method. The `Canny` function in OpenCV requires two threshold values, the lower and upper thresholds (`100` and `200` in this case), which are used to determine edge linking.

The function returns two images: `edges`, which contains the edges detected by the manual method, and `edges_opencv`, which contains the edges detected by OpenCV's Canny detector.

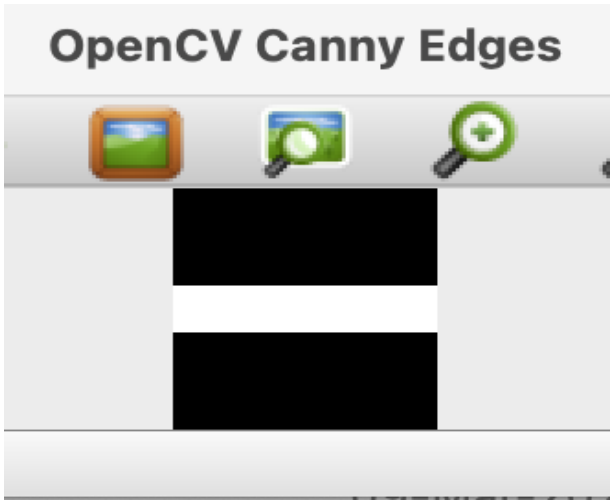


Figure 1 Edge detected by built in function

```
array([[  0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0],
       [255, 255, 255, 255, 255],
       [  0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0]], dtype=uint8)
```

Figure 2 matrix genereated by the built in function for the edge in 5*5 patch

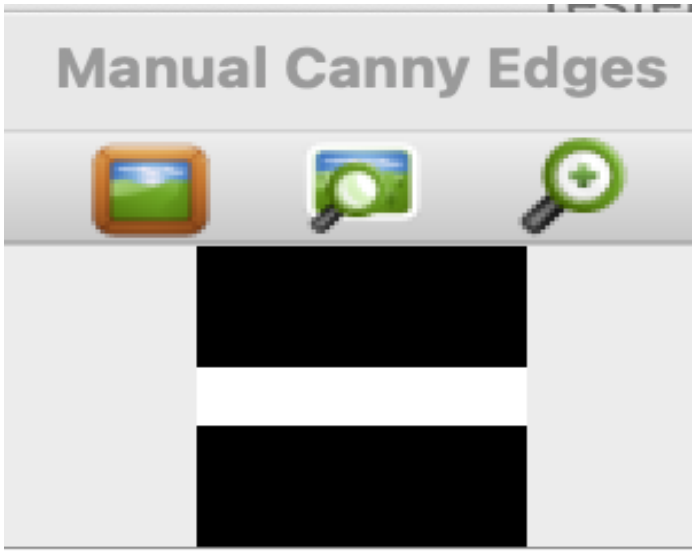


Figure 3 Edge detected manually

```
array([[  0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.],
       [255., 255., 255., 255., 255.],
       [  0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.]])
```

Figure 4 Matrix generated by manual edge detection function in a 5*5 patch

A2.

To implement the Harris corner detection algorithm manually on a 5x5 patch of an image, I used the following process

1. **Convert the image to grayscale:**
   The input color image is converted to a grayscale image since corner detection processes typically work on single-channel images.

2. **Extract the 5x5 patch from the ROI:**
   The function extracts a 5x5 pixel patch from the grayscale image. The location of the patch is determined by `patch_center` within the specified `roi`.

3. **Convert patch to float32:**
   The pixel values of the patch are converted to `float32`. This is likely for precision during the gradient calculations that follow.

4. **Calculate gradient matrices:**
   The gradients of the patch in both the x (Ix) and y (Iy) directions are calculated using the Sobel operator in python.

5. **Compute products of gradients**
   The products of gradients are calculated, which are used in the Harris corner detection algorithm to form a structure tensor for each pixel.

6. **Apply Gaussian filter to gradient products:**
   The gradient products are each convolved with a Gaussian filter to smooth them. This step helps to reduce noise and potential false corner detections.

7**. Calculate Harris response for each pixel in the patch:**
   The Harris corner response `R` for each pixel is calculated using the determinant (`detM`) and trace (`traceM`) of the structure tensor. The parameter `k` is empirically determined and affects the sensitivity of the corner detection.

8. **Threshold to identify corners:**
   A threshold is applied to the Harris response to identify strong corners. If the Harris response `R` for a pixel is greater than this threshold, it's considered a corner, and the corresponding position in `corner_response` is marked.

9. **Compare with OpenCV's Harris corner detection:**

   The region of interest from the original grayscale image is processed using OpenCV's built-in `cornerHarris` function. The result `dist` is dilated to make the corner points more visible. The corners detected by OpenCV are then thresholded and marked in `corners_opencv`.

10. **Return Results:**
   The function returns two 5x5 arrays: `corner_response` (the manually computed corners) and `corners_opencv` (the corners found by OpenCV)

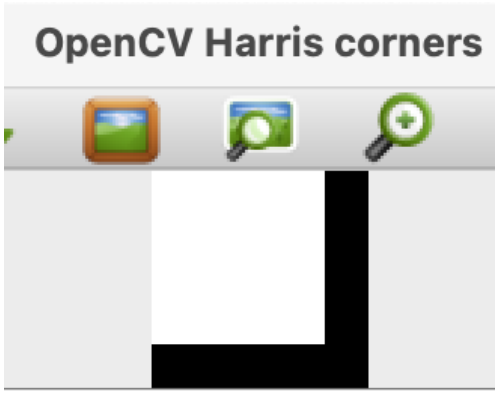


Figure 5 Edge detected by built in function in OpenCV

```
array([[255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [  0.,   0.,   0.,   0.,   0.]], dtype=float32)
```

*Figure 6 Matrix of edge detected by built in function in a 5*5 patch*

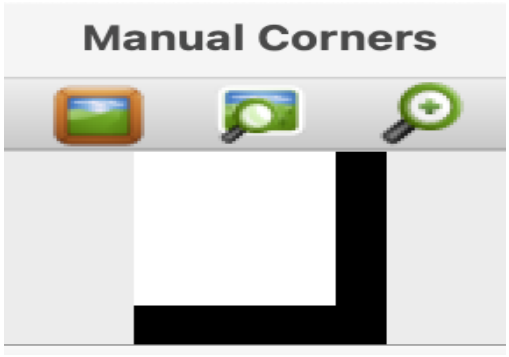


Figure 7 Edge detected by manual corner detection function

```
array([[255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [255., 255., 255., 255.,   0.],
       [  0.,   0.,   0.,   0.,   0.]])
```

*Figure 8 Matrix of edge detected by manual corner detection function in a 5*5 patch*

A3.

a) Computing the SIFT feature for each of these 2 patches and sum of squared difference (SSD)

The process to compute SIFT features and calculating sum of squared distance from 2 frames, consist of the following functions.

1. **extract_frames function**:
    - **Purpose**: To extract two specific frames from a given video file.
    - **Process**:
        - Open the video file using cv2.VideoCapture.
        - Initialize an empty list frames to store the extracted frames.
        - Loop through the frames in the video sequentially, checking each frame's index (frame_idx) to see if it matches the start_frame or end_frame.
        - If the index matches, append the current frame to the frames list.
        - Stop the loop either when the video ends (ret is False) or when two frames have been extracted.
        - Release the video capture object and return the two extracted frames.
2. **compute_sift_descriptors function**:
    - **Purpose**: To detect keypoints in an image and compute their SIFT descriptors.
    - **Process**:
        - Initialize a SIFT feature detector with specified parameters (e.g., nfeatures, contrastThreshold, edgeThreshold).
        - Detect keypoints and compute the SIFT descriptors for the entire image using sift.detectAndCompute(image, None).
        - Return the detected keypoints and their corresponding descriptors.
3. **find_super_patches function**:
    - **Purpose**: To identify a set number of prominent keypoints in an image and enlarge them to the size of 'super-patches'.
    - **Process**:
        - Initialize a SIFT feature detector.
        - Detect keypoints throughout the entire image.
        - Select a specified number of the most significant keypoints (based on their response value).
        - Convert these keypoints into 'super-patches' by enlarging their size to a specified patch_size.
        - Return the enlarged keypoints.

4. **match_keypoints function**:
    - function to match keypoints between two sets of SIFT descriptors using OpenCV's Brute Force matcher with L2 norm.
    - The matcher uses the crossCheck flag for better matching quality. The matches are sorted based on their distance (the lower, the better the match).
    - Returns the sorted list of matches.

5. **calculate_ssd function**:
    - **Purpose**: To calculate the sum of squared differences between two SIFT descriptors.
    - **Process**:
        - Take two descriptors as input.
        - Perform element-wise subtraction between the two arrays of descriptors, square the differences, and sum all the squared elements.
        - Return the scalar sum, which is the SSD value.

This process picks a super pixel on the frame and the corresponding super pixel on frame 2 and calculates its SIFT features and finally outputs the sum of the squared distance between any to descriptors.

```
if matches:
    best_match = matches[0]
    ssd = calculate_ssd(descriptors1[best_match.queryIdx], descriptors2[best_match.trainIdx])
    print(f"SSD for the best match: {ssd}")
else:
    print("No matches found")
```

```
SSD for the best match: 837.0
```

Figure 9 Code block with the final SSD output

b) Computing the Homography matrix between these two images

To compute the Homography matrix my function does the following things:

1. **Ensuring descriptors are in the correct type:**
    - The descriptors extracted from SIFT are converted into a float32 numpy array, which is the expected type for FLANN-based matching.
2. **Initializing FLANN parameters and matcher:**
    - FLANN (Fast Library for Approximate Nearest Neighbors) is used for efficient matching of SIFT descriptors between the two images.
    - A kd-tree index is created for the descriptors with a specified number of trees, which affects the speed and accuracy of the search.
    - The search_params with a specified number of checks control the accuracy of the search. A higher number of checks increases the accuracy but also the computational cost.
3. **Performing matching with FLANN:**
    - The knnMatch method finds the k-nearest neighbors for each descriptor, where k is set to 2 in this case.
    - This means for each descriptor in the first set, the two closest descriptors in the second set are found.
4. **Apply the ratio test with additional checks:**
    - The ratio test is used to retain good matches. For each pair of matches, if the distance of the closest match is less than 70% of the second-closest match, it's considered a good match. This filters out matches where there isn't a distinct nearest neighbor, which are likely to be incorrect.

- It ensures that there are at least 2 matches to perform the test (as len(match) == 2) and that the number of good matches is sufficient to compute the homography.

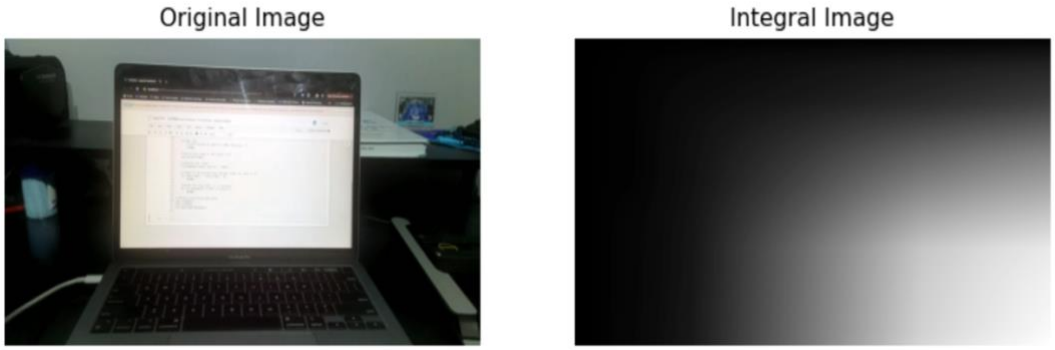
5. **Computing the homography matrix:**
    - If there are enough good matches (at least 4), the corresponding points (src_pts and dst_pts) are extracted from the keypoints of both images.
    - cv2.findHomography is used to calculate the homography matrix H using RANSAC (Random Sample Consensus), which is a robust method to estimate parameters while dealing with outliers.
    - RANSAC iteratively selects random subsets of matches and attempts to compute the homography, eventually selecting the homography that best fits the majority of matches. The threshold value of 5.0 determines how close points must be to a line to be considered as inliers.
6. **Compute the inverse of the Homography matrix:**
    - Once a homography matrix is computed, its inverse can also be calculated using np.linalg.inv(H).
    - The homography matrix maps points from the first image to the second image, while the inverse homography matrix will map points from the second image back to the first.

```
Homography matrix:
[[ 1.15833834e+00 -1.21988193e-01 -5.68751894e+02]
 [ 5.31652895e-02  9.94244571e-01  6.71452143e+01]
 [ 6.63134866e-05 -9.66056599e-05  1.00000000e+00]]
Inverse Homography matrix:
[[ 8.30105033e-01  1.46765534e-01  4.62269206e+02]
 [-4.04070763e-02  9.92125240e-01 -8.95980631e+01]
 [-5.89507112e-05  8.61123793e-05  9.60689637e-01]]
```

Figure 10 Computed Homography matrix and its inverse

A4. Computing the integral image –

1. **Convert Image to Grayscale**:
    - The input image (frame1) is converted to a grayscale image (gray_clr) because the integral image is typically calculated on single-channel images. This conversion simplifies the process and reduces computational complexity.
2. **Initialize Integral Image Array**:
    - An array integral_image with the same dimensions as the grayscale image is initialized to store the integral image. It's initialized to zeros and uses dtype=np.uint32 to ensure it can store large integer values, which is necessary because the values in the integral image can become quite large.
3. **Compute Integral Image**:
    - The integral image is computed in a cumulative manner. Each pixel in the integral image represents the sum of all pixels above and to the left of it, including itself.
    - The computation starts by copying the grayscale value of each pixel into the integral_image.
    - For each pixel at position (i, j), the value is incremented by the sum of all pixels above it (integral_image[i-1, j]) and to the left of it (integral_image[i, j-1]). If the pixel is not on the top row or the first column, the value at integral_image[i-1, j-1] is subtracted because it was added twice (once for the row above and once for the column to the left).
    - This process is carried out for each pixel, moving row by row, left to right within each row.
4. **Display Original and Integral Image**:
    - The original image (frame1) and the computed integral image (integral_image) are displayed side by side for comparison.
    - The imshow function from matplotlib is used to display the images. The integral image is displayed using a grayscale colormap (cmap='gray'), which is suitable for visualizing single-channel images.



5. **Real-time implementation:**
    - This is also implemented in real-time for the web application using the same logic as above
    - The real-time function takes frames in real-time and calculates the integral image for each frame and displays it side by side.

A5. Panorama

1. **Setup and Frame Extraction**:
    - The code begins by initializing a VideoCapture object to read from a video file.
    - It reads all frames from the video in a loop and stores them in the frames list.
2. **Initialize Panorama**:
    - The first frame of the video is used as the initial reference for the panorama. SIFT keypoints and descriptors are computed for this frame.
3. **Iterative Stitching**:
    - The code iterates through the remaining frames. For every next frame,it attempts to stitch the frame to the current panorama. This subsampling is likely an attempt to reduce computational load and memory usage.
4. **Feature Extraction**:
    - For each selected frame, SIFT keypoints and descriptors are extracted using find_keypoints_and_descriptors.
5. **Feature Matching**:
    - Features between the current panorama (last frame stitched) and the new frame are matched using a FLANN-based matcher in match_features. FLANN is used for its efficiency in handling large datasets.
6. **Filtering Matches**:
    - filter_matches applies Lowe's ratio test to filter out weak matches, keeping only the strong matches that are likely to represent correct correspondences.
7. **Compute Homography**:
    - For frames with a sufficient number of good matches (> 10), the homography matrix (M) is computed using find_homography. This matrix represents the transformation between the matched features in the current panorama and the new frame.
8. **Stitching**:
    - stitch_images applies the homography to transform the new frame and stitch it onto the current panorama. It handles the translation of the stitched image to ensure all content is visible within the output image's bounds.
9. **Display and Update**:
    - The updated panorama is displayed in a window titled 'Panoramic View'. The last frame (current panorama) and its keypoints and descriptors are updated for the next iteration.
10. **Exit Condition**:
    - The stitching process can be exited by pressing 'q'.
11. **Real-time:**
    - This function is used to convert all the frames of a video into a panorama in real-time for the web-application.
12. **Note regarding blending:**
    - I tried to blend the image while stitching them but it did not give good results so I skipped that part.

Read Me

This is a web application to perform 3 tasks
1. Measure the dimensions of an object in real-time
2. Calculate the integral image in real time
3. Stitch images to form a panorama in real time

All the python file should be in the same directory as the static and template folder and the app.py file need to be executed to run this application on a web server.

THE "**Ruchir Namjoshi Module 2 code.ipynb**" IPYNB files consists of solutions to all the 5 questions in the assignment. Just run the file with the 10-second video in the same directory.

Github link - https://github.com/ruchirnamjoshi/module-2-CV--