COMP425 Assignment - 2

Lucía Cordero Sánchez
l_corde@live.concordia.ca
40291279

March 18, 2024

1 Summary

This assignment was divided in two parts. 1) Implementation of the Hough tranform , 2) Implementation of RANSAC and homography matrix.

2 Part 1: Hough transform

2.1 Load the image, convert to grey-scale and run a canny edge detector to find edges

First, I loaded the input image using *cv2.imread*. I converted the image to grayscale using *cv2.cvtColor*. Finally, I used the Canny edge detector to find edge points using *feature.canny* from skimage.

The result was the following (the same as marked in the instructions):

Figure 1: Edges after running a detector in the input image)

2.2 Multiply the mask with the edge map within the ROI

For my implementation, the following steps were followed:

- **Get image dimensions**: Get the height and width of the edges image using edges.shape. This will be used to create the mask for the region of interest (ROI).

- **Create a binary mask for ROI**: Use the *create_mask* function to create a binary mask that represents the ROI. The *create_mask* function takes the height and width of the image as input and returns a binary mask.
- **Print and visualize the mask**: Print the mask to the console and visualize it using *plt.imshow*. This step helped me verify that the mask is correctly created and covers the desired region. This was the result for my mask (as expected):

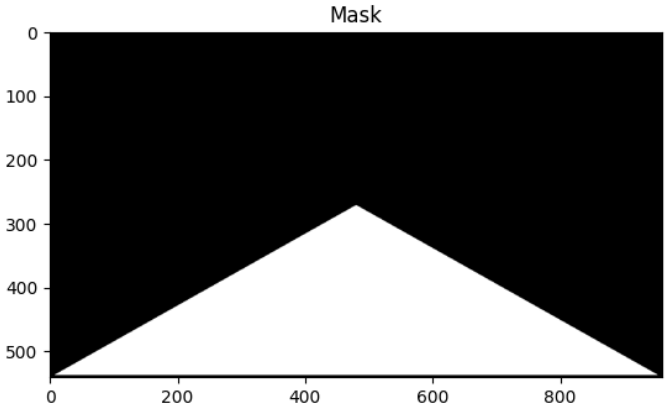

Figure 2: Created mask (as expected)

- **Convert edges and mask to uint8**: Convert the edges and mask images to uint8 data type. This step ensures that the images are in the correct format for bitwise operations.
- **Extract edge points in ROI**: I used *cv2.bitwise_and* to extract the edge points within the ROI. This is done by multiplying the edges image with the mask, effectively masking out the edges outside the ROI.
- **Visualize edges within ROI**: These were the resulting edges within the ROI, for which I used again *plt.imshow*. This step helped me see the edges that will be used for the Hough transform.

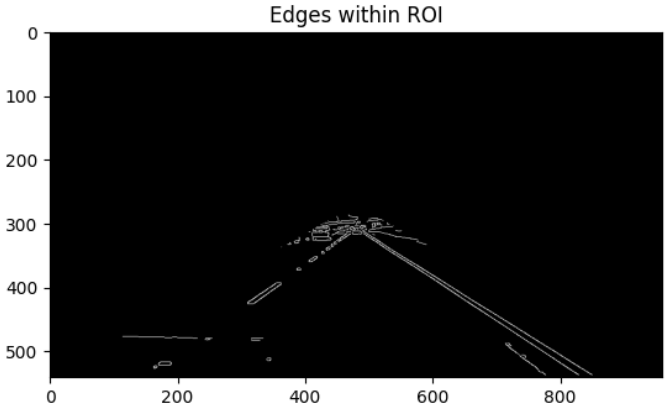

Figure 3: Map containing edges in the ROI

2.3 Implement the Hough transform within two major lanes. Use polar representation as the parameter space.

I have followed these steps to implement the Hough trasform and to visualize the major lanes in the road:

1. **Calculate diagonal length**: I calculated the diagonal length of the edges image to ensure that the accumulator array covers all possible rho values. This length is used to create the range of *rho* values.

2. **Generate theta values**: Generate a range of theta values from -90 degrees to 90 degrees in radians. These values represent the possible angles of lines in the image.

3. **Initialize accumulator**: Create an accumulator array to store the Hough transform values. The accumulator has dimensions based on the ranges of *rho* and *theta* values.

4. **Iterate over edge pixels**: Iterate over the edge pixels in the edges image. For each edge pixel, iterate over all *theta* values and calculate the corresponding *rho* value using the Hough transform equation:
$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

5. **Increment the accumulator**: Increment the accumulator at the index corresponding to the calculated *rho* and *theta* values.

6. **Find major lane**: I found the peak in the accumulator to determine the major lane. This is done by finding the maximum value in the accumulator and getting the corresponding rho and theta values.

7. **Zero out neighborhood**: My code zeroes out the values in the accumulator around the neighborhood of the peak to avoid detecting the same lane again.

8. **Find second lane**: Find the peak in the accumulator again to determine the second lane (in this case, the left lane), after zeroing out the neighborhood.

9. **Create lines**: Used the *create_line* function to create lines corresponding to the detected rho and theta values for both the major and second lanes.

10. **Plot the results**: Plotted the original image and overlay the detected lanes on top of it using different colors. The result was the following:

Figure 4: Plotting the Hough transform results

The parameters were selected based on the specific requirements and characteristics of the Hough transform and lane detection task. Here's an explanation of why these values were chosen:

1. **neighborhood_size** = 500: This parameter determines the size of the neighborhood around the peak in the accumulator that is zeroed out. By setting it to 500, I effectively removed the influence of nearby peaks in the accumulator, which helps in detecting distinct lanes. The actual value of 500 was chosen based on the expected width of the lanes and the density of edge points in the image.

   When the lanes in the image are relatively wide, a larger *neighborhood_size* may be appropriate to ensure that the algorithm does not mistakenly detect other parts of the lane as a separate peak. Plus, a larger value can also help in reducing the sensitivity to noise in the edge detection process.

   Ultimately, the choice depends on a trade-off between sensitivity to noise, robustness to lane width variations, and computational efficiency. Experimenting with different values of neighborhood_size and evaluating the performance of the lane detection algorithm, this was the most suitable parameter value.

3 Part 2: RANSAC, Homography

3.1 Complete the matchPics (I1, I2) function to run SIFT on two images and find candidate matching between them

Here's an explanation of the parameters and choices in my *matchPics* function:

1. **Grayscale Conversion**: The function first checks if the input images I1 and I2 are in color or grayscale format. SIFT works on grayscale images, so if the input images are in color, they are converted to grayscale using the *rgb2gray* function. This step ensures that the SIFT algorithm can be applied correctly.

2. **SIFT Initialization**: The function initializes the SIFT detector using the SIFT class. This step creates an instance of the SIFT detector that will be used to detect keypoints and compute descriptors for the images.

3. **Key Point Detection and Descriptor Extraction**: For each image, the function detects keypoints using the *detect_and_extract* method (from skimage) of the SIFT detector. It

then computes descriptors for these keypoints using the descriptors attribute of the SIFT detector. The locations of the keypoints are stored in the locs1 and locs2 arrays for the respective images.

4. **Matching Descriptors**: The function matches the descriptors between the two images using the *match_descriptors* function. The ***max_ratio*** parameter is used to filter out ambiguous matches. This parameter specifies the maximum allowed ratio between the best match and the second-best match. Matches with ratios greater than *max_ratio* are considered ambiguous and are discarded. By choosing an appropriate max_ratio value, we improve the accuracy of the matching process by reducing the number of false matches.

5. **Output**: The function returns three outputs: matches, locs1, and locs2. The matches array contains the indices of matched keypoints between I1 and I2. Each row of matches corresponds to a matched keypoint pair, where the first element is the index of the keypoint in I1 and the second element is the index of the keypoint in I2. The locs1 and locs2 arrays contain the locations of keypoints in the respective images. This was the plotted result, very similar to the expected one (I rotated the cover to have a view on the similarity with respect to the proposed solution, for practical purposes).

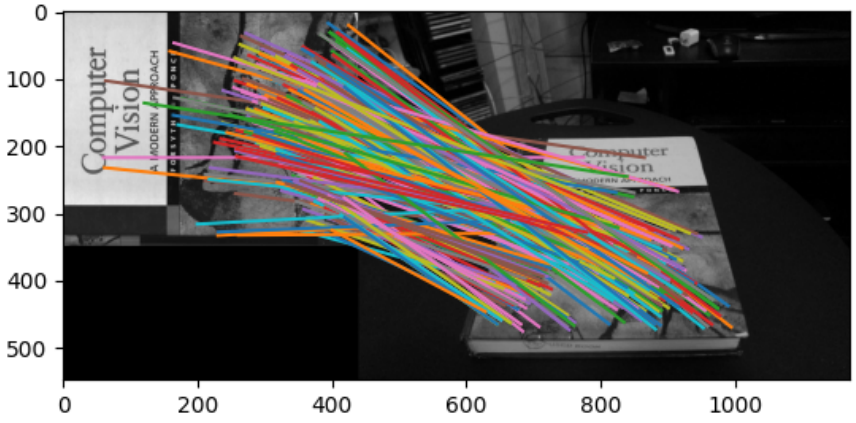

Figure 5: Plotting the result before RANSAC

6. **Choice of max_ratio**: A higher max_ratio value allows for more leniency in the matching process, potentially including more matches but also increasing the risk of including false matches. On the other hand, a lower max_ratio value reduces the risk of false matches but may also exclude some valid matches. The value of **0.6** is a commonly used default value that balances these trade-offs and is often found to work well in practice.

3.2 Complete the computeH_ransac (matches, locs1, locs2) function to get a result only with inlier matches, and create the bounding box

In this section, I will cover the key aspects of my implementation in this function: first, **I defined two new functions**, *dist* and *computeHomography*.

The ***computeHomography*** function computes the homography matrix given corresponding points in two images. It takes two arrays of shape (N, 2) representing (x, y) coordinates in the first and second images. It constructs the matrix A using the corresponding points and performs Singular Value Decomposition (SVD) on A. The last column of V from SVD is reshaped into a 3x3 matrix to obtain the homography matrix H. Finally, H is normalized by dividing by its last element to ensure it is a valid homography matrix. The function returns the computed homography matrix H.

The ***dist*** function calculates the geometric distance between two corresponding points given the homography matrix H. It takes two arrays of shape (2,) representing (x, y) coordinates of a point in the first and second images, respectively, and the homography matrix H. It first converts the points to homogeneous coordinates. It then estimates the transformed point in the second image using H. The Euclidean distance between the estimated point and the actual corresponding point in the second image is calculated and returned.

After this brief introduction, these are the key aspects of my coding journey to complete the *computeH_ransac* method.

1. **Iterations and Threshold**: The function uses RANSAC (Random Sample Consensus) to estimate the homography between the two sets of keypoints (locs1 and locs2). RANSAC is an iterative method that randomly samples a subset of the matches, computes a model (in this case, a homography), and then checks how well the model fits the remaining matches. The num_iterations parameter specifies the number of iterations RANSAC should perform, and the threshold parameter defines the maximum allowable distance between a match and its corresponding point under the model for it to be considered an inlier.

2. **Random Selection of Matches**: In each iteration, the function randomly selects **4 matches** (*selected_matches*) without replacement using *np.random.choice*. It then retrieves the corresponding keypoints from locs1 and locs2 for these matches (*selected_locs1* and *selected_locs2*).

3. **Computing Homography**: The function computes the homography H using the *computeHomography* function (that I defined previously) with the selected keypoints.

4. **Calculating Inliers**: For each match, it calculates the distance (using *dist* function) between the transformed keypoint (locs1[i] transformed by H) and the corresponding keypoint in locs2. If this distance is less than the threshold, the match is considered an inlier and added to the inliers array.

5. **Updating Best Model**: The function updates the best_inliers array if the current set of inliers is larger than the previously best set of inliers.

6. **Final Homography Calculation**: After all iterations, the function calculates the final homography *bestH* using the keypoints corresponding to the best set of inliers (locs1[matches[best_inliers, 0]] and locs2[matches[best_inliers, 1]]). This step ensures that the homography is calculated using the best set of matches found by RANSAC.

7. **Output (matrix estimation and plot)**: The function returns the final homography *bestH* and the indices of the best set of inliers *best_inliers*. This is the value of the estimated homography matrix *bestH* after the computations:

$$\begin{bmatrix} 0.742492 & -0.343798 & 237.968397 \\ -0.001235 & 0.227920 & 191.191202 \\ -0.000007 & -0.000918 & 1.000000 \end{bmatrix}$$

The plot after the final homography is very similar to the expected result from the PDF:

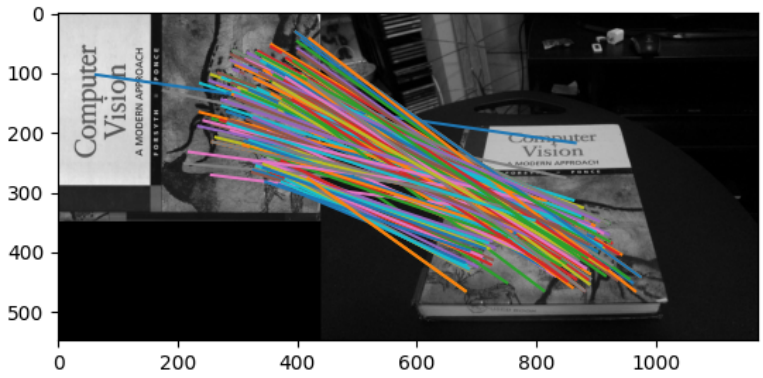

Figure 6: Result without outliers (after RANSAC)

8. Choice of Parameters: The choice of ***num_iterations*** and threshold depends on the the characteristics of the images. A higher *num_iterations* value increases the likelihood of finding a good model but also increases computation time. The threshold value determines the tolerance for considering a match an inlier and should be chosen based on the expected noise level in the matches. For this case, I used **1500** iterations (which resulted in a robust result yet computationally not very expensive).

Once I had my homography matrix, I run the already implemented function to apply the homography on the coordinates of the bounding box, resulting in the following plot:

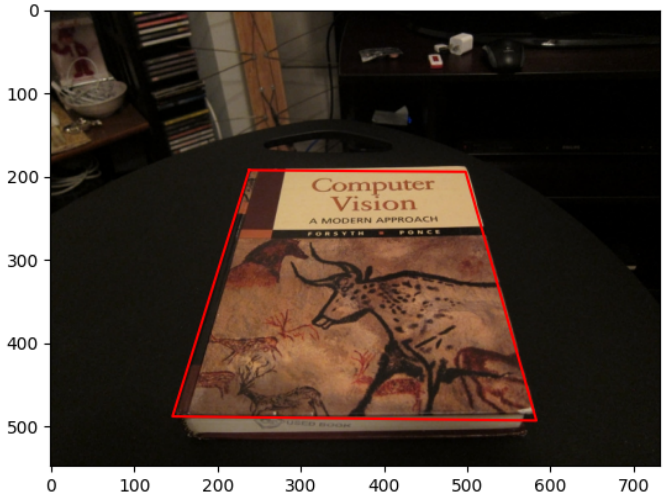

Figure 7: Bounding box for book cover

3.3 Complete the compositeH (H, template, img) function to warp a template image (in this case, the Harry Potter cover) with a target image.

Here's a detailed explanation of my function:

1. **Creating a Mask**: The function first creates a mask of the same size as the template image. This mask is initialized with ones, indicating that the entire template image will be used in the composite.

2. **Warping the Mask**: Later, it warps the mask using the inverse of the homography matrix H and the warp function. This step ensures that the mask is transformed correctly to align with the target image.

3. **Warping the Harry Potter cover template**: Similarly, the function warps the HP template image using the inverse of the homography matrix H and the warp function. By performing this, it aligns the template image with the target image based on the estimated homography.

4. **Combining Images**: The function then creates a copy of the target image (img) called composite_img. It uses the warped mask to selectively combine the warped template image (warped_template) with the target image. Pixels where the mask is greater than 0 indicate areas where the warped template should be placed on top of the target image. These areas in the composite_img are replaced with the corresponding pixels from the warped template.

5. **Final result**: The function outputs the composite image composite_img, which is the target image with the warped template image overlaid on top of it based on the estimated homography. In our case, the final result was this plot:

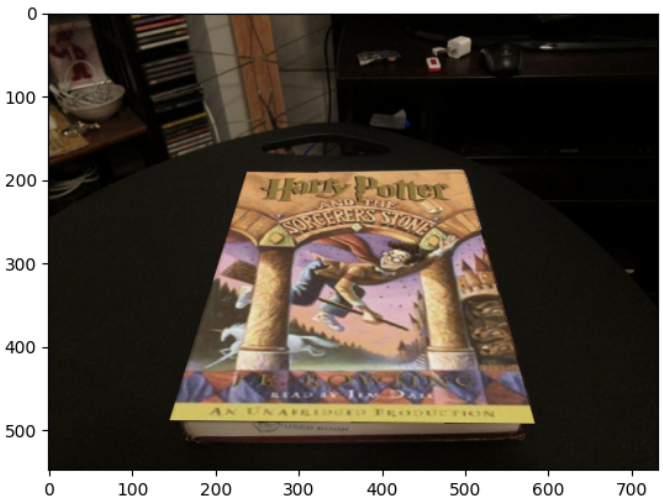

Figure 8: Final result after warping the template to the image