Assignment 2 CV

Name	ID
Nadine Mohamed Tamish	8092
Hour Mohamed AbdElaziz	8119
Seif Maged Farouk	8387

Augmented Reality with Planar Homographies: Detailed Function Analysis

Overview

This report comprehensively analyzes a Python notebook implementing augmented reality using planar homographies. The notebook demonstrates how to detect a book cover in video frames, compute homography transformations, and project virtual content onto the book cover.

Function Analysis

1. `convert_to_grayscale(image)`

- **Purpose**: Converts a color image to grayscale
- **Parameters**:
  - `image`: Input color image in BGR format (OpenCV's default color order)
- **Implementation Details**:
  - Uses OpenCV's `cvtColor` function, which changes the color space of an image
  - `COLOR_BGR2GRAY` is the conversion code that specifies conversion from BGR to Grayscale
- **Why It's Used**: Grayscale images are often used in computer vision algorithms because they simplify processing by reducing dimensionality and focusing on intensity patterns

2. `load_first_frame(video_path)`

- **Purpose**: Extracts only the first frame from a video file
- **Parameters**:
  - `video_path`: String containing the file path to the video
- **Implementation Details**:
  - `VideoCapture(video_path)`: Creates an object to read from video files or capture from cameras
  - `cap.read()`: Returns two values:
    - `ret`: Boolean indicating if frame was successfully read
    - `frame`: The actual image data (if successful)
  - `cap.release()`: Closes the video file or capturing device
  - Error handling raises a descriptive ValueError if frame reading fails
- **Why It's Used**: When you only need a single sample frame from a video for initial testing

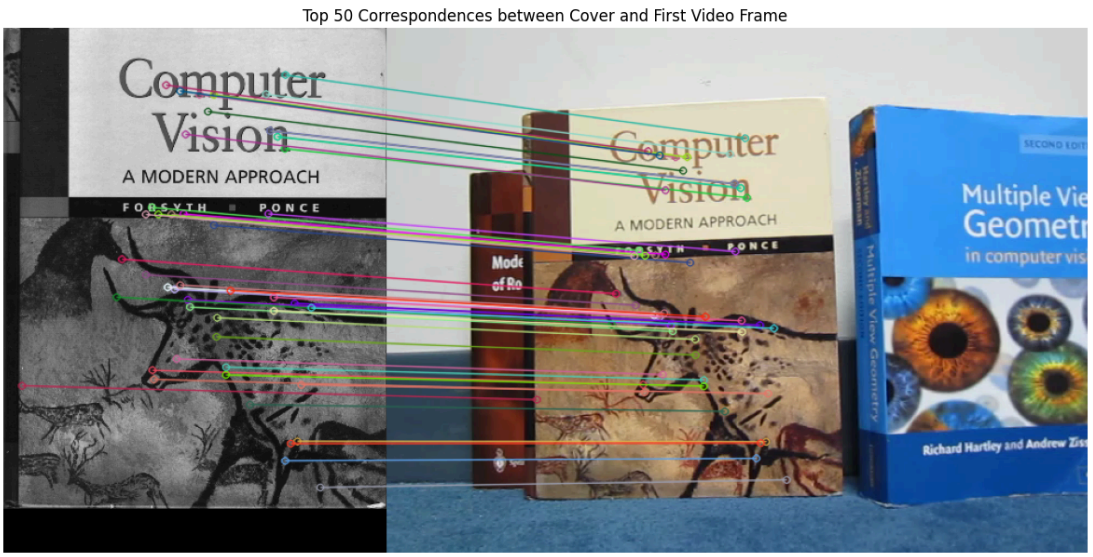
3. `load_video_frames(video_path)`

- **Purpose**: Loads all frames from a video file into memory
- **Parameters**:
  - `video_path`: String containing the file path to the video
- **Implementation Details**:
  - Creates an empty list of `frames` to store each video frame
  - Uses a `while` loop to read frames until `cap.read()` returns `ret=False`
  - Each successfully read frame is appended to the frames list
  - `cap.release()` frees resources after reading

- **Why It's Used**: When you need to process every frame in a video and have sufficient memory to store them all

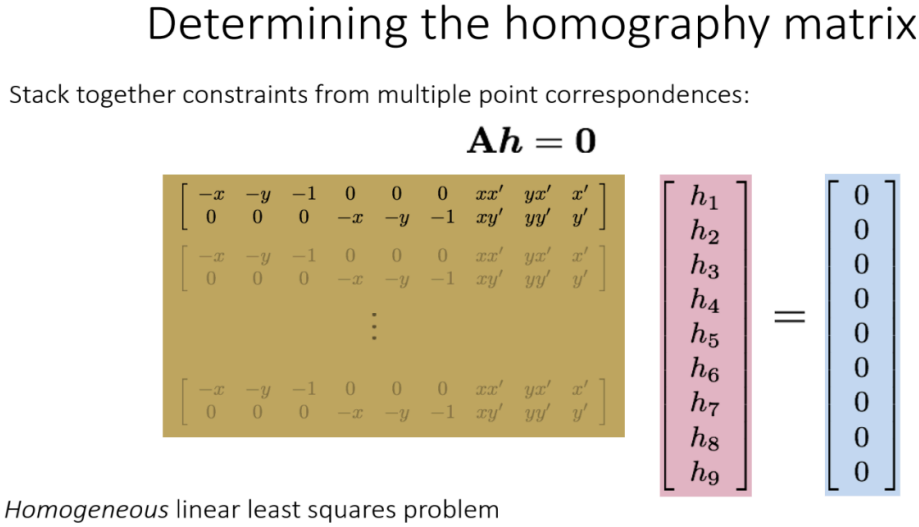
4. `get_corresponding_points(image1, image2)`

- **Purpose**: Identifies matching points between two images using SIFT features
- **Parameters**:
  - `Image 1`: First input image (grayscale)
  - `Image 2`: Second input image (grayscale)
- **Implementation Details**:
  - `cv2.SIFT_create()`: Creates a SIFT (Scale-Invariant Feature Transform) detector object
  - `sift.detectAndCompute()`:
    - Detects keypoints (distinctive points in the image)
    - Computes descriptors (feature vectors that describe each keypoint)
    - `No` parameter means no mask is used (process the entire image)
  - `cv2.BFMatcher(normType=cv2.NORM_L2)`:
    - Creates a brute-force matcher
    - `normType=cv2.NORM_L2` specifies Euclidean distance for comparing descriptors
  - `bf.knnMatch(descriptors1, descriptors2, k=2)`:
    - Performs k-nearest neighbor matching
    - `k=2` finds the two closest matches for each descriptor in descriptors1
  - Lowe's ratio test:
    - Filters matches based on the ratio of distances between the best and second-best match
    - `0.75` is a common threshold that filters out many false matches
  - `sorted(good_matches, key=lambda x: x.distance)`:
    - Sorts matches by distance (smaller distance = better match)
  - `good_matches[:50]`: Takes only the 50 best matches
- **Why It's Used**: Feature matching is crucial for finding corresponding points between images, which is the foundation for computing homographies



5. `compute_homography_dlt(src_pts, dst_pts)`

- **Purpose**: Computes the homography matrix using the Direct Linear Transform (DLT) algorithm
- **Parameters**:
  - `src_pts`: Source points (coordinates in the first image)
  - `dst_pts`: Destination points (corresponding coordinates in the second image)
- **Implementation Details**:
  - Constructs the DLT equation system:
    - For each point pair, extract coordinates `(x,y)` from the source and `(u,v)` from the destination
    - Creates two rows in matrix A for each point correspondence
    - These rows enforce the constraint that H transforms source points to destination points
  - `np.linalg.svd(A)`:
    - Performs Singular Value Decomposition on matrix A
    - Returns three components: U, S, and Vt (transpose of V)
    - `_` indicates unused variables (U and S are not needed)
  - `Vt[-1]`:
    - Takes the last row of Vt (equivalent to last column of V)
    - This corresponds to the eigenvector with the smallest eigenvalue
    - In DLT, this represents the least-squares solution to the system
  - `reshape(3, 3)`: Converts the vector into a 3×3 matrix
  - `H /= H[2, 2]`: Normalizes H by dividing by its last element (conventional choice)
- **Why It's Used**: The DLT algorithm provides a direct method to compute the homography from point correspondences by solving a linear system



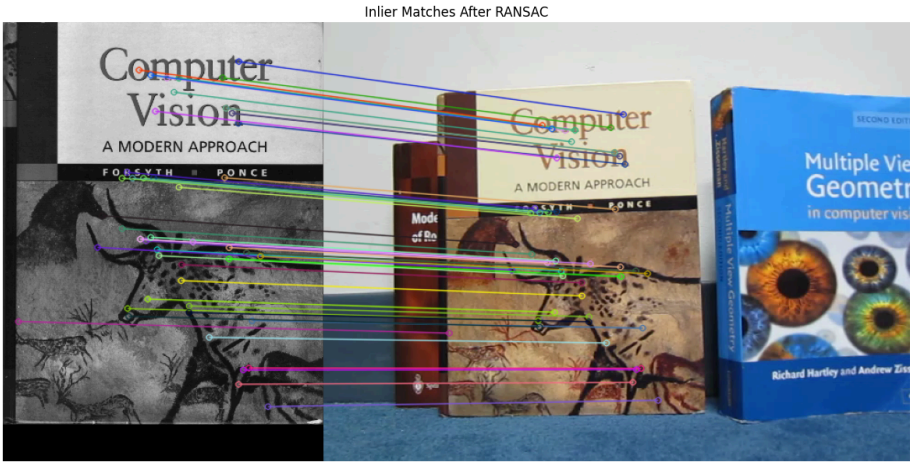

```
H using our function:

[[ 7.73175741e-01  2.75722138e-03  1.19569964e+02]

 [-5.11103701e-02  7.78602149e-01  7.74981421e+01]

 [-9.11956235e-05 -7.43420663e-05  1.00000000e+00]]
```

```
H using built-in function:

[[ 7.73918433e-01  2.98716605e-03  1.19494352e+02]

 [-5.06993013e-02  7.79124943e-01  7.74114010e+01]

 [-8.98341594e-05 -7.36718754e-05  1.00000000e+00]]
```

6. `ransac(src_pts, dst_pts, threshold=5.0, num_iterations=1000, early_stopping=True)`

- **Purpose**: Implements the RANSAC algorithm for robust homography estimation
- **Parameters**:
  - `src_pts`: Source points (Nx2 array) - Point coordinates from the reference image
  - `dst_pts`: Destination points (Nx2 array) - Corresponding point coordinates in the target image
  - `threshold`: Maximum pixel distance for a point to be considered an inlier (default: 5.0)
  - `num_iterations`: Maximum number of RANSAC iterations (default: 1000)
  - `early_stopping`: Boolean to enable early termination if a high inlier count is found (default: True)
- **Implementation Details**:
  - Initialization:
    - `best_inliers`: Tracks which points are inliers for the best model
    - `best_inliers_count`: Tracks how many inliers the best model has
    - `best_H`: Stores the best homography matrix found
  - For each RANSAC iteration:
    - `np.random.choice(n_points, 4, replace=False)`:
      - Randomly selects 4 point pairs (minimum needed for homography)
      - `replace=False` ensures no repeats
    - Calls `compute_homography_dlt()` to calculate H from these 4 points
    - Projection steps:
      - `np.hstack((src_pts, np.ones((n_points, 1))))`: Converts to homogeneous coordinates
      - `(H @ src_h.T).T`: Applies homography to all points
      - `projected /= projected[:, 2].reshape(-1, 1)`: Normalizes homogeneous coordinates
    - Error calculation:
      - `np.linalg.norm(dst_pts - projected[:,:2], axis=1)`: Computes Euclidean distance between projected points and actual destination points
      - `errors < threshold`: Creates a boolean mask where errors are below threshold
      - `np.sum(inliers)`: Counts total inliers
    - Model update:
      - Updates the best model if more inliers are found
      - Early stopping if 95% of points are inliers (optional)
  - Final refinement:
    - Recomputes the homography using all inliers for better accuracy
    - Requires at least 4 inliers (minimum for homography computation)

- **Why It's Used**: RANSAC provides robust estimation in the presence of outliers, which is critical when working with real-world feature matches that contain incorrect correspondences



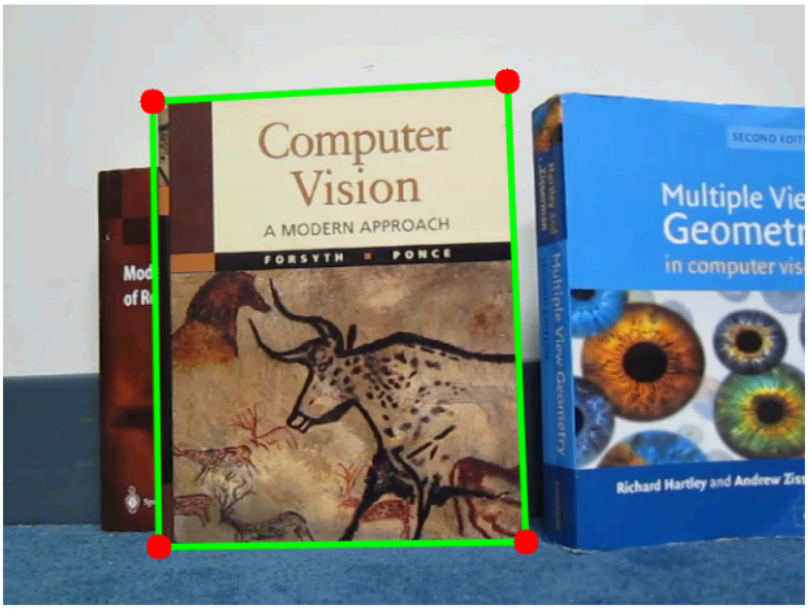
7. Computing Book Corner Coordinates

- **Purpose**: Computes where the corners of the book cover appear in the video frame
- **Implementation Details**:
  - `cv_cover.shape`: Gets height (h) and width (w) of the book cover image
  - Creates an array with the four corners of the book cover: top-left, top-right, bottom-right, bottom-left
  - `reshape(-1, 1, 2)`: Reshapes for compatibility with `perspectiveTransform`
  - `cv2.perspectiveTransform(book_corners, H)`: Applies the homography transformation to all corners at once

8. Drawing Projected Rectangle

- **Purpose**: Visualizes the projected book outline on the video frame
- **Implementation Details**:
  - `cv2.polylines()`:
    - Draws connected line segments
    - `isClosed=True`: Connects the last point to the first
    - `color=(0, 255, 0)`: Green color in BGR format
    - `thickness=3`: Line width of 3 pixels
    - `lineType=cv2.LINE_AA`: Anti-aliased line for smoother appearance
  - `cv2.circle()`:
    - Draws circles at each corner
    - `(int(x), int(y))`: Center coordinates (converted to integers)
    - `10`: Radius of 10 pixels
    - `(0, 0, 255)`: Red color in BGR format
    - `-1`: Filled circle

Book Corners with Projection



8. `remove_black_bars(frame)`

- **Purpose**: Automatically detects and crops black borders from a video frame
- **Parameters**:
  - `frame`: Input video frame (color image)
- **Implementation Details**:
  - `cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)`: Converts the frame to grayscale
  - `cv2.threshold()`:
    - Applies a binary threshold to the grayscale image
    - `10`: Threshold value (pixels with intensity below 10 are considered black)
    - `255`: Maximum value to assign to pixels above threshold
    - `cv2.THRESH_BINARY`: Thresholding type (simple binary threshold)
  - `cv2.findNonZero(thresh)`: Finds all non-zero (white) pixels in the thresholded image
  - `cv2.boundingRect(coords)`: Computes the bounding rectangle of all non-zero pixels
    - Returns `x, y`: Top-left corner coordinates
    - Returns `w, h`: Width and height of the rectangle
  - `frame[y:y+h, x:x+w]`: Crops the original frame to the bounding rectangle
- **Why It's Used**: Removes unnecessary black borders from frames, which can occur in videos with different aspect ratios, maximizing the usable area

9. `crop_ar_video(frame, book_corners)`

- **Purpose**: Crops and resizes a video frame to match the book cover's dimensions and aspect ratio
- **Parameters**:
  - `frame`: Input video frame (color image)
  - `book_corners`: Coordinates of the four corners of the book cover

- **Implementation Details**:
    - First removes black bars using the `remove_black_bars()` function
    - Reshapes `book_corners` into a standard format (-1, 2)
    - Extracts x and y coordinates from book corners
    - Calculates:
        - `book_width`: Distance between leftmost and rightmost corners
        - `book_height`: Distance between topmost and bottommost corners
        - `book_aspect_ratio`: Width/height ratio of the book cover
        - `frame_aspect_ratio`: Width/height ratio of the current frame
    - Aspect ratio comparison and cropping:
        - If the frame is wider than book, crops the sides (left and right)
        - If the frame is taller than book, crops the top and bottom
    - `cv2.resize(cropped, (book_width, book_height))`: Resizes the cropped frame to exactly match the book dimensions
- **Returns**:
    - `resized`: The cropped and resized frame
    - `book_width`: Width of the book cover
    - `book_height`: Height of the book cover
- **Why It's Used**: Ensures the AR content matches the book cover's dimensions, providing a realistic overlay effect

10. Warping and Overlaying AR Content

- **Purpose**: Applies the AR content to the book cover in the video frame
- **Implementation Details**:
    - Gets the dimensions of the target video frame
    - Creates a copy of the original frame to overlay content on
    - `cv2.warpPerspective()`:

- Applies perspective transformation to the cropped AR image
- `H`: The homography matrix
- `(book_video_w, book_video_h)`: Output image dimensions
- `dst=overlayed`: Directly modifies the overlayed image
- `borderMode=cv2.BORDER_TRANSPARENT`: Preserves transparency for smooth blending

11. `create_ar_application()`

- **Purpose**: Creates an augmented reality application by processing each frame of the input videos
- **Parameters**: None (uses global variables `book_frames`, `ar_source_frames`, and `cv_cover`)
- **Implementation Details**:
  - Preprocessing:
    1. Converts all book frames to grayscale and stores them in a list
    2. Initializes `final_video_frames` list to store the augmented frames
  - For each pair of frames (limited to the smaller video's length):
    1. Feature matching:
       - Finds corresponding points between the book cover and the current video frame
       - Skips the frame if fewer than 4 matches are found
    2. Homography estimation:
       - Extracts matching points and reshapes them
       - Computes homography using RANSAC
       - Skips the frame if no valid homography is found
       - Refines homography using only inliers
    3. AR frame preparation:
       - Defines the book cover corners in original coordinates
       - Crops and resizes the AR frame to match the book dimensions
    4. Warping and blending:

        - Creates a copy of the current book frame as the base
        - Warps the AR frame onto the book position using a perspective transform
        - Checks if the warped frame is valid
    5. Visualization:
        - Projects the book corners using the homography
        - Draws a green outline around the detected book
    6. Stores the result in the output list
- **Returns**: List of augmented video frames
- **Why It's Used**: Core function that implements the entire AR workflow frame by frame

12. `save_ar_video(frames, output_path, fps=25)`

- **Purpose**: Saves processed frames as a video file
- **Parameters**:
    - `frames`: List of image frames to be saved as a video
    - `output_path`: Path where the video file will be saved
    - `fps`: Frames per second (default: 25)
- **Implementation Details**:
    - Error handling for empty frames list
    - Gets dimensions from the first frame to set the video size
    - `cv2.VideoWriter_fourcc(*'XVID')`: Creates a FourCC code for the XVID codec
    - `cv2.VideoWriter()`:
        - Creates a video writer object
        - `output_path`: Path to the output file
        - `fourcc`: Video codec
        - `fps`: Frame rate
        - `(width, height)`: Frame dimensions
    - Writes each frame to the video file in a loop
    - Releases the video writer to close the file properly
- **Why It's Used**: Exports the processed AR frames as a playable video file

Conclusion

This notebook demonstrates a complete augmented reality pipeline using planar homographies:

1. **Feature Detection and Matching**: Using SIFT to find robust correspondences between the book cover and video frames
2. **Homography Estimation**: Computing the transformation matrix with RANSAC to handle outliers
3. **Content Preparation**: Cropping and resizing AR content to match the book cover dimensions
4. **Warping and Blending**: Applying the AR content to the detected book in each video frame
5. **Video Output**: Creating a final video with the augmented reality effect

Image Mosaics

2.1 Getting correspondences and computing homography

Same as part 1

2.2 Warping between image planes

Functions implemented:

- **homography(xy_tuple, H) :** tuple of a point is passed and the homography matrix, used to apply homography transformation for each point

- **delete_black(image):** image is passed to delete black holes in transformed image

- **get_subpixel(img, x, y):** image is passed and a point (x and y), gets the 4 surrounding pixels of the passed x and y.

- **bilinear_interpolation(x, y, subpixels):** a function that returns the equation of the bilinear interpolation

- **get_bound_box(img, H):** The image and homography matrix are passed. Nested loops apply a homography transformation to all of the image's pixels and get the bounding box coordinates that can hold the warped images.

- **warp_image(img, H):** compute bounding box coordinates
  Calculate the coordinates of the new image
  `accumulator`: keeps track of how many values have been added to each pixel, per channel
  After looping on each channel independently, each pixel is processed multiple times, so it's normalized to compute the output as the average

Output of warp image for the first image:

- **inverse_warp_image(img, img_warped, bounding_box, H):** fills black holes in warped image using inverse warping and bilinear interpolation

Output of inverse warp for image 1

2.3 Output Mosaic

Function implemented:

- **stitch_image(image1_warped, image2, bounding_box)**
  Warped image 1 and image 2 are passed and fit in the bounding box created in 2.2
  Image 2 is shifted with dimensions relative to the bounding box to be aligned with image 1
  Black canvas is created to lay the stitched images on it
  A loop is created to process each pixel in image 2 to shift them to be aligned with image 1
  Delete_black function is called to delete black holes