University POLITEHNICA of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Fundamentals of Image Processing and Computer Vision

Assignment 2

Student: Angheluță Andrei-Alexandru
Group: 441G
Coordonator: Dr. Ing. Claudia Cristina Oprea

2025

PANORAMA STITCHING

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

picture_a = cv2.imread(r"C:\Users\Alex\IPIVA\LAB\Teme\Tema 2\set2\24.png")
picture_b = cv2.imread(r"C:\Users\Alex\IPIVA\LAB\Teme\Tema 2\set2\11.png")
picture_c = cv2.imread(r"C:\Users\Alex\IPIVA\LAB\Teme\Tema 2\set2\42.png")

im = [picture_a, picture_b, picture_c]
im_gray = [cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) for img in im]

plt.figure(figsize=(10, 6))
plt.title('Images')
plt.subplot( *args: 1, 3, 1)
plt.imshow(cv2.cvtColor(picture_a, cv2.COLOR_BGR2RGB))
plt.title('Image 1')
plt.subplot( *args: 1, 3, 2)
plt.imshow(cv2.cvtColor(picture_b, cv2.COLOR_BGR2RGB))
plt.title('Image 2')
plt.subplot( *args: 1, 3, 3)
plt.imshow(cv2.cvtColor(picture_c, cv2.COLOR_BGR2RGB))
plt.title('Image 3')
plt.show()

MAX_FEATURES = 750
GOOD_MATCH_PERCENT = 0.20
orb = cv2.ORB_create(MAX_FEATURES)
```

```python
kp1, d1 = orb.detectAndCompute(im_gray[0], None)
kp2, d2 = orb.detectAndCompute(im_gray[1], None)
kp3, d3 = orb.detectAndCompute(im_gray[2], None)

matcher = cv2.DescriptorMatcher_create(cv2.DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING)

matches32 = matcher.match(d3, d2, None)
matches32 = sorted(matches32, key=lambda x: x.distance)
numGoodMatches32 = int(len(matches32) * GOOD_MATCH_PERCENT)
good_matches = matches32[:numGoodMatches32]
imMatches32 = cv2.drawMatches(im_gray[2], kp3, im_gray[1], kp2, good_matches, outImg: None)

matches21 = matcher.match(d2, d1, None)
matches21 = sorted(matches21, key=lambda x: x.distance)
numGoodMatches21 = int(len(matches21) * GOOD_MATCH_PERCENT)
good_matches = matches21[:numGoodMatches21]
imMatches21 = cv2.drawMatches(im_gray[1], kp2, im_gray[0], kp1, good_matches, outImg: None)

plt.figure()
plt.subplot( *args: 1, 2, 1)
plt.imshow(imMatches21[:, :, ::-1])
plt.title('Match 1-2')
plt.subplot( *args: 1, 2, 2)
plt.imshow(imMatches32[:, :, ::-1])
plt.title('Match 2-3')
plt.show()
```

```python
def keypoints_des(image):  2 usages
    orb = cv2.ORB_create()
    keypoints, descriptors = orb.detectAndCompute(image, None)
    return keypoints, descriptors


def match_keypoints(descriptors1, descriptors2, good_match_percent=0.17):  1 usage
    matcher = cv2.BFMatcher( *args: cv2.NORM_HAMMING, crossCheck=True)
    matches = matcher.match(descriptors1, descriptors2)
    matches = sorted(matches, key=lambda x: x.distance)
    num_good_matches = int(len(matches) * good_match_percent)
    good_matches = matches[:num_good_matches]
    return good_matches


def stitch_images(image1, image2, keypoints1, keypoints2, matches):  1 usage
    points1 = np.float32([keypoints1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
    points2 = np.float32([keypoints2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
    homography, mask = cv2.findHomography(points2, points1, cv2.RANSAC, ransacReprojThreshold: 5.0)
    if homography is None:
        print("The homography could not be calculated.")
        return None

    height1, width1, _ = image1.shape
    height2, width2, _ = image2.shape
    panorama_width = width1 + width2
    panorama_height = max(height1, height2)
```

```
    result = cv2.warpPerspective(image2, homography, dsize: (panorama_width, panorama_height))
    result[0:height1, 0:width1] = image1

    gray_result = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
    ret, binary_threshold = cv2.threshold(gray_result, thresh: 1, maxval: 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(binary_threshold, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    x, y, w, h = cv2.boundingRect(max(contours, key=cv2.contourArea))
    cropped_result = result[y:y + h, x:x + w]
    return cropped_result


im = [picture_a, picture_b, picture_c]
im_gray = [cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) for img in im]

keypoints_list = []
descriptors_list = []
for gray_image in im_gray:
    keypoints, descriptors = keypoints_des(gray_image)
    keypoints_list.append(keypoints)
    descriptors_list.append(descriptors)
stitched_image = im[0]
current_keypoints = keypoints_list[0]
current_descriptors = descriptors_list[0]

for i in range(1, len(im)):
    matches = match_keypoints(current_descriptors, descriptors_list[i])
    stitched_result = stitch_images(stitched_image, im[i], current_keypoints, keypoints_list[i], matches)

    if stitched_result is not None:
        stitched_image = stitched_result

        plt.figure(figsize=(10, 10))
        plt.imshow(stitched_image[..., ::-1])
        plt.title(f'Stitched Image {i}')
        plt.axis('off')
        plt.show()

        current_keypoints, current_descriptors = keypoints_des(cv2.cvtColor(stitched_image, cv2.COLOR_BGR2GRAY))

gray_panorama = cv2.cvtColor(stitched_image, cv2.COLOR_BGR2GRAY)
ret, binary_threshold = cv2.threshold(gray_panorama, thresh: 1, maxval: 255, cv2.THRESH_BINARY)
contours, hier = cv2.findContours(binary_threshold, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

max_contour_area = 0
largest_contour = None
for contour in contours:
    contour_area = cv2.contourArea(contour)
    if contour_area > max_contour_area:
        max_contour_area = contour_area
        largest_contour = contour

x, y, width, height = cv2.boundingRect(largest_contour)
new = stitched_image[y:y + height, x:x + width]

plt.figure()
plt.imshow(new[..., ::-1])
plt.title('Final Panorama')
plt.show()
```

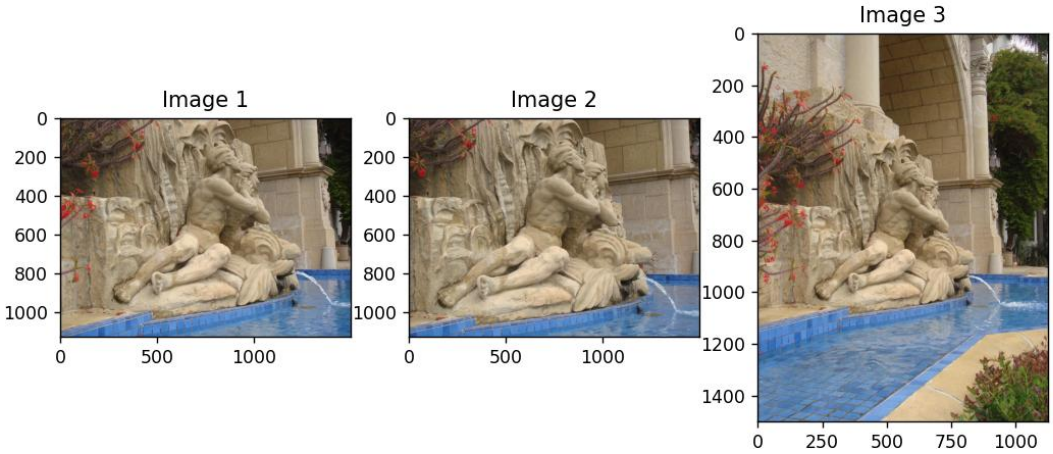
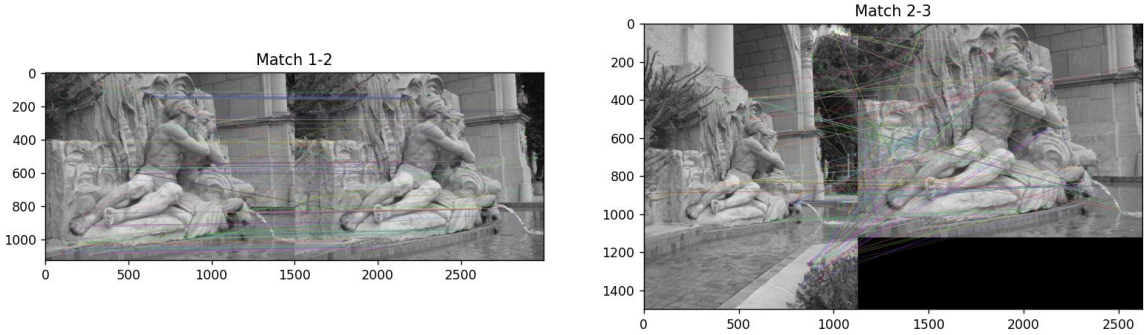
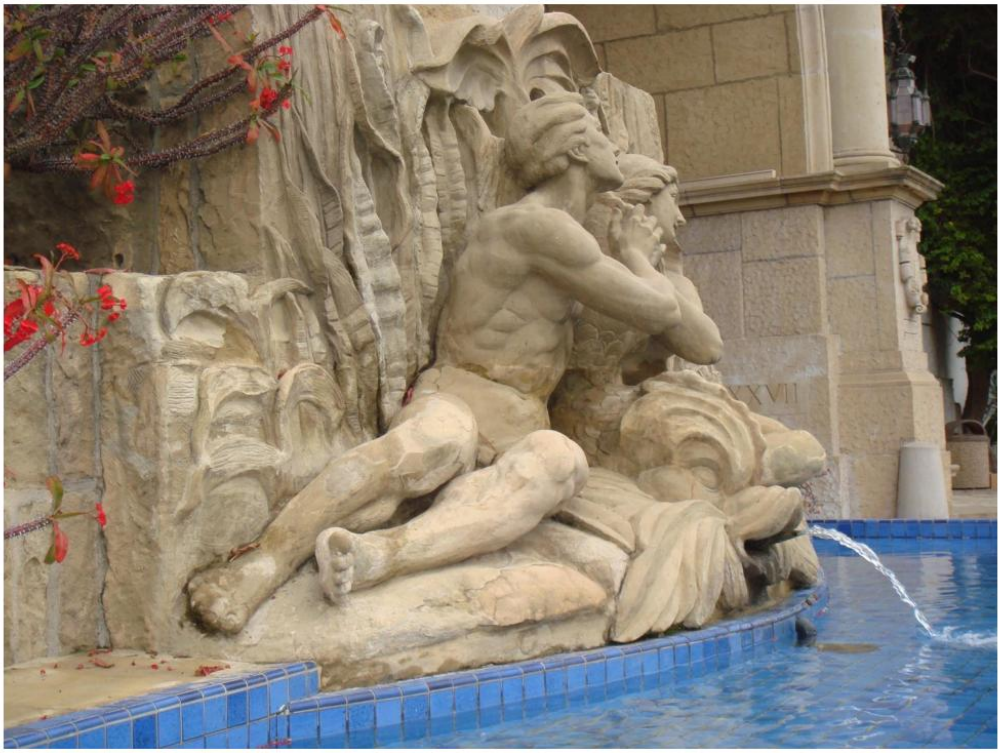


Image 1
Image 2
Image 3


Match 1-2
Match 2-3

Stitched Image 1



Stitched Image 2

Explanation:

The code above is focused on panorama stitching using the ORB algorithm to detect keypoints and compute descriptors.

First, we read the three images (24.png, 11.png, 42.png) using *cv2.imread* and store them in a list. Then, all images are converted to grayscale, because ORB feature detection and descriptor computation work on intensity information, not color. After that, we display the three original images side-by-side, just to confirm the input order and check that the dataset is loaded correctly.

Next, we initialize ORB with *MAX_FEATURES = 750*. This value controls the maximum number of keypoints ORB will try to detect in each image. After that, ORB keypoints and descriptors are computed for each grayscale image using *orb.detectAndCompute(...)*. The descriptors are binary vectors that describe the local neighborhood around each keypoint and allow matching between images.

After descriptors are computed, we perform a quick visual check of matching for the pairs (2–1) and (3–2). For matching, we use a brute-force Hamming matcher, using *cv2.DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING*, which is appropriate for ORB's binary descriptors. The matches are sorted by distance (smaller distance means a better match), and for this visual check we keep the best 20% using *GOOD_MATCH_PERCENT = 0.20*; we later keep the default 17% of matches inside *match_keypoints()* and use them to estimate the homography. Then, we draw the matches

using *cv2.drawMatches* and display them. This step is useful to see if the images have enough overlap and if the correspondences look fine before stitching.

To automate the stitching process, we define three helper functions:

1. keypoints_des(image)

   This function runs ORB again on a given grayscale image and returns its keypoints and descriptors. Inside this function we create a new ORB instance with default settings and recompute features on the current panorama.

2. match_keypoints(descriptors1, descriptors2, good_match_percent=0.17)

   This function matches descriptors using *BFMatcher* with Hamming distance and *crossCheck=True*. Cross-checking means a match is accepted only if both descriptors consider each other as the best match. Matches are sorted by distance, and only a percentage of the best matches is kept. Here the default is 0.17, meaning the best 17% of matches are used for homography.

3. stitch_images(image1, image2, keypoints1, keypoints2, matches)

   This is the main stitching step. It extracts the matched keypoint coordinates from both images and estimates a homography matrix using *cv2.findHomography(..., cv2.RANSAC, 5.0)*. RANSAC helps reject wrong matches (outliers) so that the transformation is computed from consistent correspondences.
   Then, the second image is warped into the first image's coordinate system using *cv2.warpPerspective*. A larger canvas is created using *panorama_width = width1 + width2* and *panorama_height = max(height1, height2)*, and the first image is copied onto the warped result.
   Finally, the function removes black borders by thresholding the result, finding external contours, selecting the largest contour, and cropping to its bounding rectangle.

After defining these functions, we start the panorama construction. We initialize the panorama with the first image (*stitched_image = im[0]*) and store the current keypoints and descriptors for that image. Then we iterate through the remaining images (image 2, then image 3). For each new image we:

- find matches between the current panorama descriptors and the next image descriptors;
- stitch the new image onto the panorama using the computed homography;
- display the intermediate stitched panorama;
- recompute keypoints and descriptors on the updated panorama so the next step uses features from the current combined result.

At the end, we perform a final crop on the stitched panorama. We threshold the grayscale panorama to isolate the valid region, find contours, select the largest contour, and crop to its bounding box. This ensures the final panorama has clean borders without the empty black

areas. The final panorama is displayed, although a small black region can remain depending on how the warping and threshold-based cropping behave on the borders.