

✓ Panorama Stitching in OpenCV

Dr.-Ing. Antje Muntzinger, Hochschule für Technik Stuttgart

antje.muntzinger@hft-stuttgart.de

In this notebook, we will stitch a panorama from single images in OpenCV. We will explore two methods: a fast method using OpenCV's `Stitcher` class that handles most intermediate steps automatically, and a more hands-on method where we implement the different steps one by one.

```
# install required packages specified in pipfile
# !pipenv install
```

```
# imports
from matplotlib import pyplot as plt
from matplotlib import gridspec as gridspec
%matplotlib inline
```

```
import numpy as np
import cv2
```

```
from IPython.display import Image, display
```

Task 1: Image Preparation =

TODO: 1a) Load two or more overlapping images that you want to stitch. You can use the provided images, but it is highly encouraged to take some overlapping photos yourself. Note that not all photos can successfully be stitched together without modifications, so in case you encounter problems, try the provided images first. **(1 point)**

```
##### TODO: adapt n_images to your number of images, and adapt the paths below to load your images
# single images named 1.jpg, 2.jpg etc.
n_images = 3
```

```
# fill list of paths
image_paths=[]
for i in range(n_images):
    image_paths.append('images/'+str(i+1)+'.jpg')
```

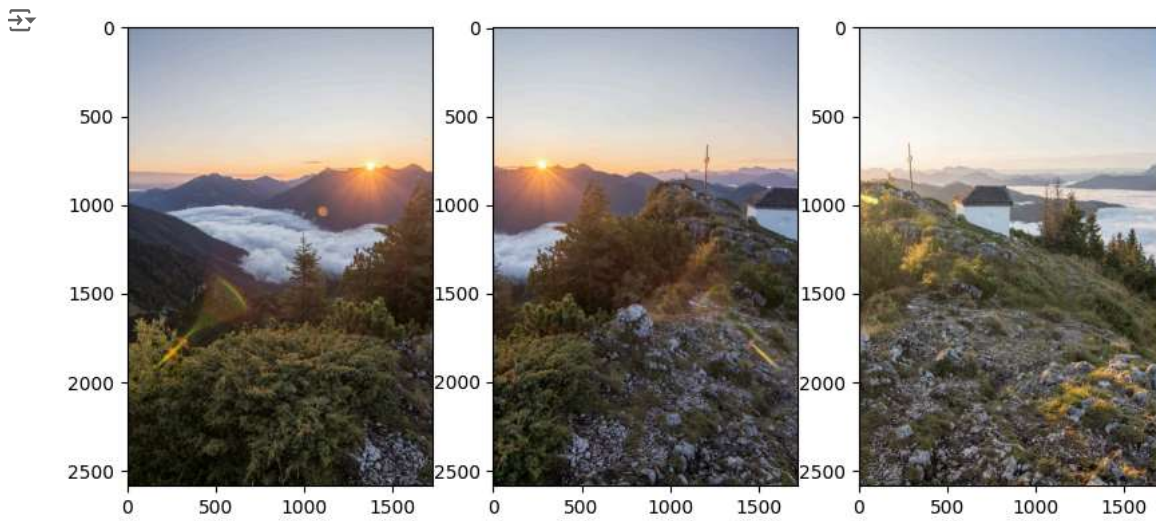
```
##### END STUDENT CODE
```

```
# fill list of images
imgs = []
for i in range(n_images):
    img = cv2.imread(image_paths[i])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # in case of memory error, potentially reduce image size
    width = int(img.shape[1]/2)
    height = int(img.shape[0]/2)
    img = cv2.resize(img, (width, height))

    imgs.append(img)
```

```
# show the original pictures
fig, axes = plt.subplots(1, n_images, figsize=(10, 30))
for i in range(n_images):
    axes[i].imshow(imgs[i])
```



✓ Panorama stitching - the fast way

Let's start by following the fast method to stitch a panorama using OpenCV's `stitcher` class. Note that depending on your choice of single images, the stitching is not always successful. Also note that due to the underlying random sampling in RANSAC, the result is not deterministic, you might get different results for multiple code runs.

```
# create stitcher object and stitch images
stitchy=cv2.Stitcher.create()
(dummy,output)=stitchy.stitch(imgs)

# check if the stitching procedure was successful
if dummy != cv2.STITCHER_OK:
    # .stitch() returns a true value if stitching is
    # done successfully
    print("Stitching was't successfull!")
else:
    print('Your panorama is ready :-)')

# final output
fig, ax = plt.subplots(1,1, figsize=(10, 30))
ax.imshow(output)
```

➡ Your panorama is ready :-)




✓ Panorama stitching - step by step

Now let's again stitch a panorama without using the `stitcher` class in order to understand what is happening under the hood. For the sake of simplicity, we only use the first two images for stitching. First we double the size of the second image to make room for the first image to be

warped into this new image. The result looks as follows:

```
# create a blank image the same size as the second image - note that an image is just a numpy array here
blank_image = np.zeros((imgs[1].shape[0], imgs[1].shape[1], 3), np.uint8)
```

```
# horizontally concatenates images of same height
imgs[1] = cv2.hconcat([blank_image, imgs[1]])
plt.imshow(imgs[1])
```

 <matplotlib.image.AxesImage at 0x7b42a96977c0>



Task 2: SIFT Descriptors =

Now we use SIFT to detect feature points and descriptors in both images and we plot the descriptors.

TODO: 2a) Use OpenCV's SIFT method to detect keypoints and descriptors of the two images. Plot the resulting keypoints and descriptors in the two images. **(4 points)**

Hint: You can use the example code from the lecture slides as reference.

```
len(imgs)
```

 3

```
##### TODO: Instantiate SIFT detector
sift = cv2.SIFT_create(contrastThreshold=0.05, edgeThreshold=0.05)

##### TODO: find the keypoints and descriptors with SIFT
keypoints1, descriptors1 = sift.detectAndCompute(imgs[0], None)
keypoints2, descriptors2 = sift.detectAndCompute(imgs[1], None)

# Function for reusability
def plot_keypoints_and_descriptor(img, keypoints, descriptors, sno):
    img_keypoints = cv2.drawKeypoints(img, keypoints, None)
    plt.figure(figsize=(8, 6))
    plt.imshow(img_keypoints)
    plt.title('Keypoints')
    plt.axis('off')
    plt.show()

# Histograms for descriptors
histograms = []
for desc in descriptors:
    histograms.append(cv2.calcHist([desc], [0], None, [256], [0, 256]))

# Plot the histograms for each image (modify for clarity)
plt.figure(figsize=(10, 5))
for i in range(n_images):
    plt.plot(histograms[i].ravel())
    plt.title('SIFT Descriptor Histogram (Image ' + str(sno) + ')')
plt.show()

##### TODO: plot result

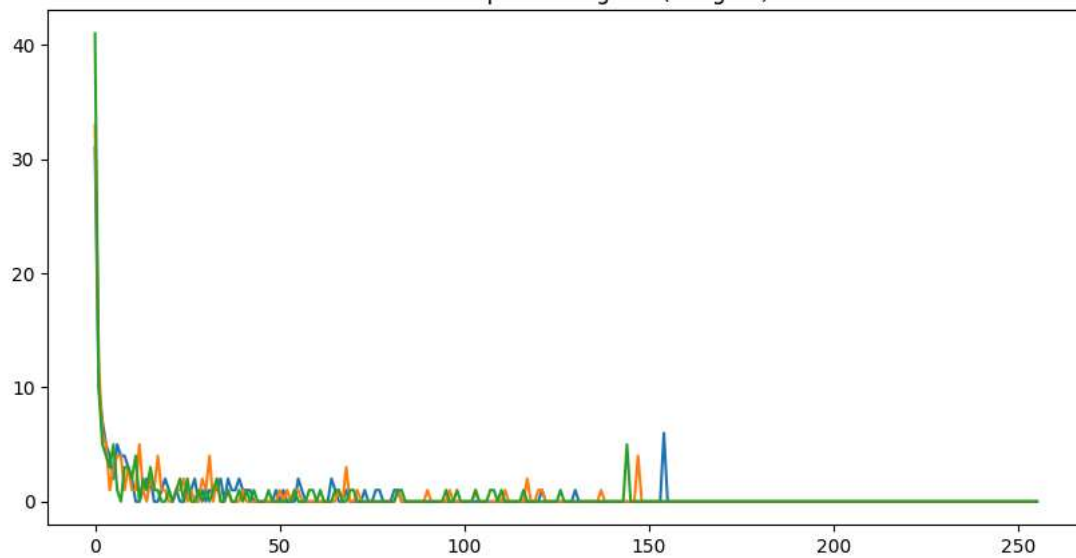
plot_keypoints_and_descriptor(imgs[0], keypoints1, descriptors1, 1)
plot_keypoints_and_descriptor(imgs[1], keypoints2, descriptors2, 2)
```



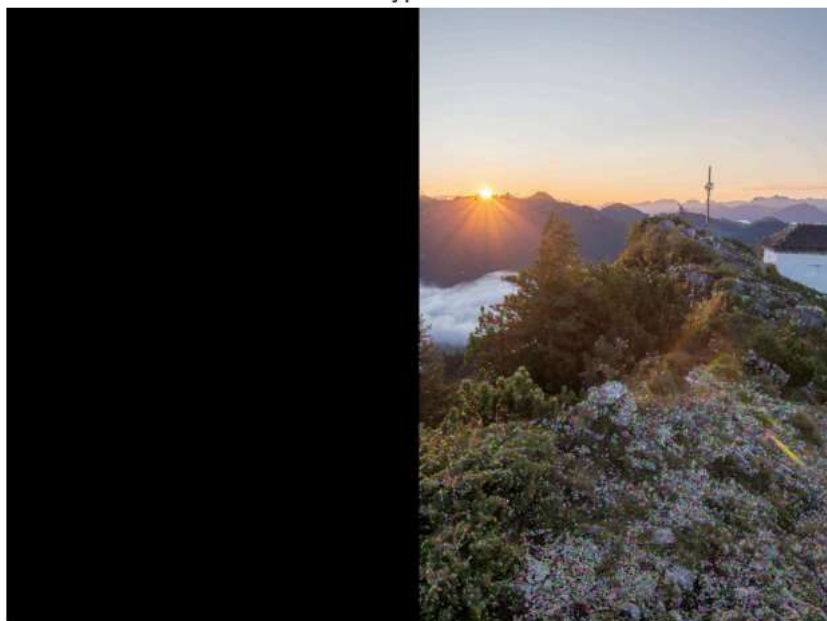
Keypoints



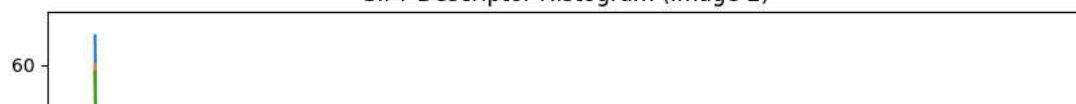
SIFT Descriptor Histogram (Image 1)

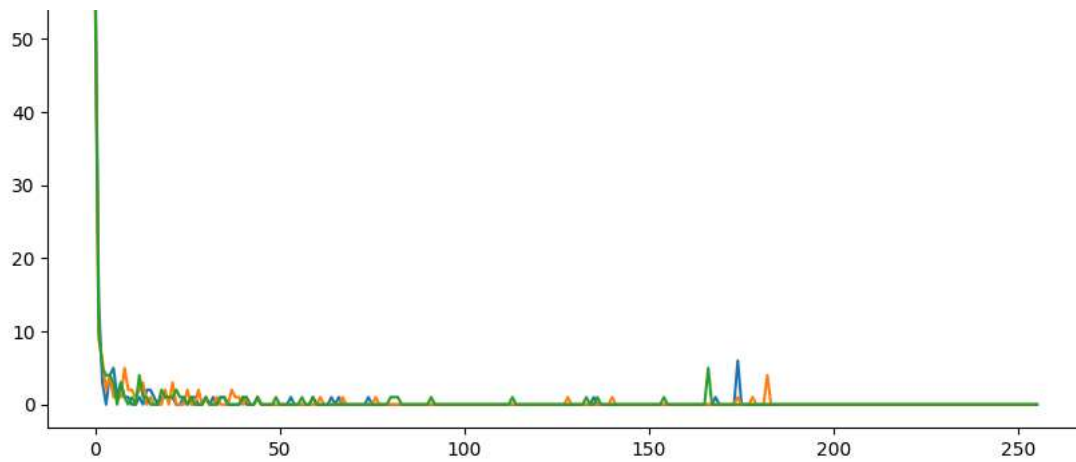


Keypoints



SIFT Descriptor Histogram (Image 2)





TODO: 2b) Why do we use SIFT feature points and not Harris Corners for matching? (2 points)

YOUR ANSWER:

- Correlation is not rotation invariant (Harris corners are rotation invariant, but we need the local surrounding as well for SIFT)
- Correlation is sensitive to photometric changes.
- Normalized correlation is sensitive to non linear photometric changes and even slight geometric distortions (e.g. small camera movements).
- Could be slow check all features against all features

Task 3: Matching Feature Points =

TODO: 3a) Find matches between the two images using FLANN (Fast Library for Approximate Nearest Neighbors). (2 points)

Hint: You can find a documentation of FLANN here: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

```
##### TODO: find matches with FLANN

# Initialize FLANN-based matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

# Match descriptors of the two images using FLANN
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

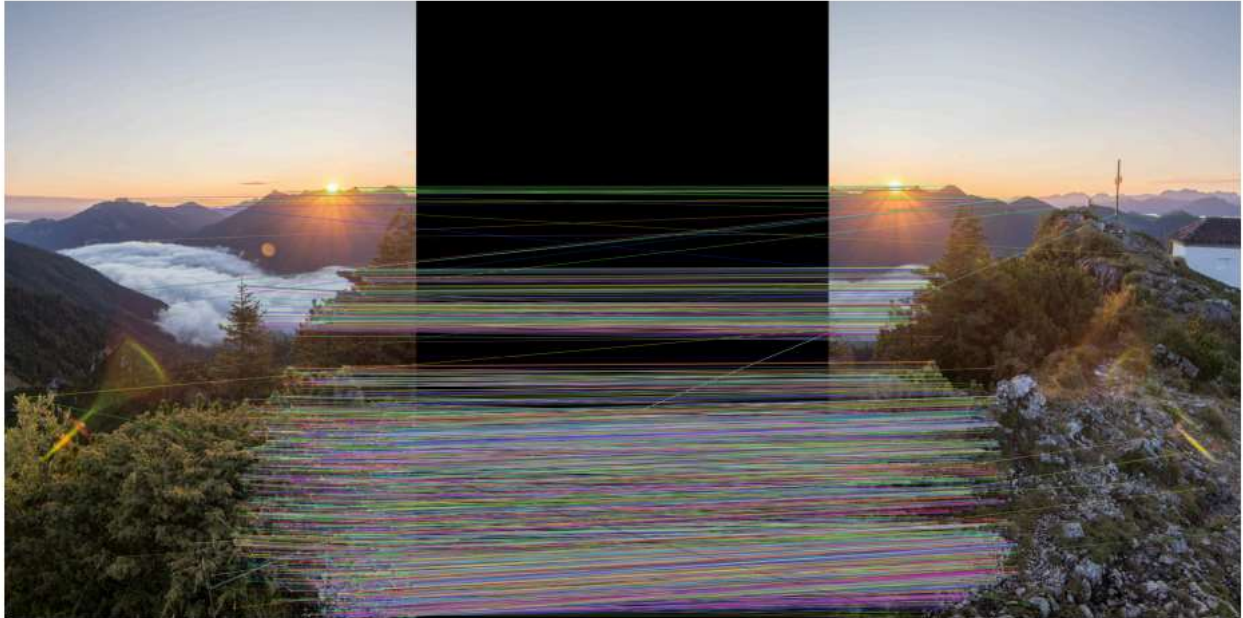
# Apply ratio test to filter good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)

# Draw matches
img_matches = cv2.drawMatches(imgs[0], keypoints1, imgs[1], keypoints2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_F

# Display the matches
plt.figure(figsize=(12, 6))
plt.imshow(img_matches)
plt.title('Feature Matches')
plt.axis('off')
plt.show()
```




Feature Matches



TODO: 3b) Store good matches following Lowe's ratio test (see lecture slides). Again, you can use the documentation linked above. (2 points)

```
##### TODO: store all the good matches in a list as per Lowe's ratio test.

# Apply Lowe's ratio test to filter good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)

# Store good matches
good_matches_points = []
for match in good_matches:
    good_matches_points.append((keypoints1[match.queryIdx].pt, keypoints2[match.trainIdx].pt))

# Convert points to numpy array
good_matches_points = np.array(good_matches_points)

# Print the number of good matches
print("Number of good matches:", len(good_matches_points))
```

Number of good matches: 2273

TODO: 3c) Why do we look at two different distances in Lowe's ratio test? Why not simply use a threshold? (2 points)

YOUR ANSWER:

The purpose of using two different distances in Lowe's ratio test is to provide a more robust criterion for selecting good matches.

Task 4: Model Fitting (RANSAC) =

TODO: 4a) Use the matches to calculate a homography between the two images. Print the homography matrix. (3 points)

Hint: You can find a tutorial on RANSAC here: https://docs.opencv.org/4.x/d1/de0/tutorial_py_feature_homography.html

```
##### TODO: YOUR CODE GOES HERE

# Convert keypoints to numpy arrays
src_pts = np.float32([keypoints1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
dst_pts = np.float32([keypoints2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)

# Calculate homography using RANSAC
homography, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Print the homography matrix
print("Homography Matrix:")
print(homography)
```



```
Homography Matrix:
[[ 2.50551199e+00  2.76401784e-01 -4.22579772e+02]
 [ 3.42038615e-01  1.64951225e+00 -5.02984953e+02]
 [ 4.31683071e-04  2.93262058e-05  1.00000000e+00]]
```

TODO: 4b) Which of the 9 values in the homography matrix was not computed by RANSAC? Where does this value come from, and could you theoretically use another number instead? **(2 points)**

YOUR ANSWER:

In the homography matrix obtained from RANSAC, eight of the nine values are computed during the RANSAC estimation process, while the ninth value is typically set to 1. This value corresponds to the scaling factor in the homography matrix.

TODO: 4c) Plot the two images and draw matches between features in the two images in red. Apply the homography to the boundary of the first image and plot the new boundary (after applying the homography) in blue into the second image. **(4 points)**

TODO: YOUR CODE GOES HERE

```
# TASK 1
## Plot the two images and draw matches between features in red
def plot_matches(img1, img2, matches, kp1, kp2):
    img_matches = cv2.drawMatches(img1, kp1, img2, kp2, matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    plt.figure(figsize=(12, 6))
    plt.imshow(img_matches)
    plt.title('Matches Between Images')
    plt.axis('off')
    plt.show()

## Plot matches between features in the two images
plot_matches(imgs[0], imgs[1], good_matches, keypoints1, keypoints2)

# TASK 2
## Apply the homography to the boundary of the first image
h, w = imgs[0].shape[:2]
pts = np.array([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]], dtype=np.float32).reshape(-1, 1, 2)
dst = cv2.perspectiveTransform(pts, homography)

## Plot the new boundary (after applying the homography) in blue onto the second image
img2_homography = cv2.polylines(imgs[1].copy(), [np.int32(dst)], True, (0, 0, 255), 3)

## Plot the result
plt.figure(figsize=(12, 6))
plt.imshow(img2_homography)
plt.title('Second Image with Homography Boundary')
plt.axis('off')
plt.show()
```



Matches Between Images



Second Image with Homography Boundary



TODO: 4d) Finally, warp the first image into the second using the homography. You can use OpenCV's `warpPerspective()` to do this - look up the interface in the online documentation. Afterwards, blend the warped first image and the second image and plot the final panorama. (2 points)

Hint: To blend the two images, you can simply use the `add()` function as learned in the lesson. However, this will cause the overlapping area to be very light, because both pixel values add up. Alternatively, you can use `addWeighted()` with a blending factor `alpha` of 0.5. See the documentation here: https://docs.opencv.org/3.4/d5/dc4/tutorial_adding_images.html.

```
##### TODO: Warp the first image using the homography
img1_warped = cv2.warpPerspective(imgs[0], homography, (imgs[1].shape[1], imgs[1].shape[0]))

##### TODO: Blend the warped image with the second image using alpha blending
alpha = 0.5
blended_image = cv2.addWeighted(img1_warped, alpha, imgs[1], 1 - alpha, 0)

##### TODO: Display the blended image
plt.figure(figsize=(12, 6))
plt.imshow(blended_image)
plt.title('Final Panorama')
plt.axis('off')
plt.show()
```




Final Panorama



That's it! You have just stitched your first panorama :-)

Task 5: Theory Questions =

TODO: 5a) You have bought a new camera lens with a focal length of 50mm for your SLR camera. You know that the distance between the lens and the camera sensor is 6cm. How far away from the lens should you place an object to be in focus in the image? Explain your result either by writing the calculations down or by text. **(2.5 points)**

YOUR ANSWER:

Given:

$$f := 0.05\text{m}$$

$$s' := 0.06\text{m}$$

Thin Lens Formula:

$$\frac{1}{f} = \frac{1}{s} + \frac{1}{s'}$$

Therefore, you should place the object approximately 30 centimeters away from the lens for it to be in focus on the camera sensor.

As the focal length remains constant, changes in object distance will result in a corresponding change in image distance to maintain focus.

Solution:

$$\frac{1}{0.05} = \frac{1}{s} + \frac{1}{0.06} \quad \rightarrow \quad s \text{ approx. } 0.03 \text{ metres}$$

TODO: 5b) You are using a camera with intrinsic camera matrix $K = \begin{pmatrix} 2642 & 0 & 1034 \\ 0 & 2642 & 764 \\ 0 & 0 & 1 \end{pmatrix}$ (values given in pixels). What are the values of your 5 intrinsic parameters? **(2.5 points)**

YOUR ANSWER:

```
display(Image('images/q2.png'))
```