```
test_each_cell = True
```

Project 1: Panorama! (Total 100 points)

The primary objective of this project is to develop a comprehensive end-to-end pipeline for panorama stitching, similar to the panorama mode available on modern smartphones. Panorama stitching is a technique used to seamlessly combine multiple photographs to create a single, wide-angle seamless image that captures a broader view than a single shot could achieve.

Each image should have few repeated local features (~30–50% or more, emperically chosen). The following method of stitching images should work for most image sets but you'll need to be creative for working on harder image sets.

From a set of narrow field of view unordered pair of images:

To a Seamless Panorama:

Overview:

Change Runtime Type to T4 GPU:

Note: *It is not required for P1 but for following projects that involve deep learning. Feel free to skip this step.*

1. Click on "Runtime" in the menu bar.
2. Select "Change runtime type".
3. Choose "T4" from the "Hardware accelerator" dropdown menu.
4. Click "Save".

```
# Make Sure NVIDIA T4 GPU is connected:

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
/bin/bash: line 1: nvidia-smi: command not found
```

Part A: Input Images

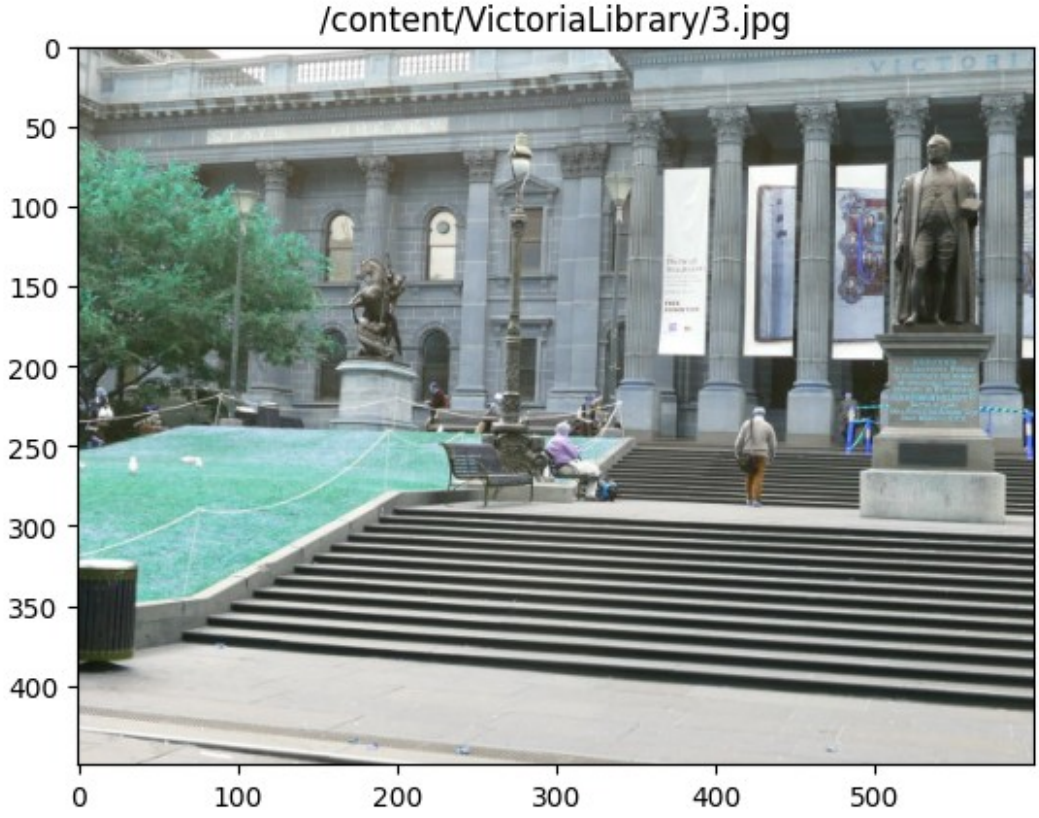
Overview:

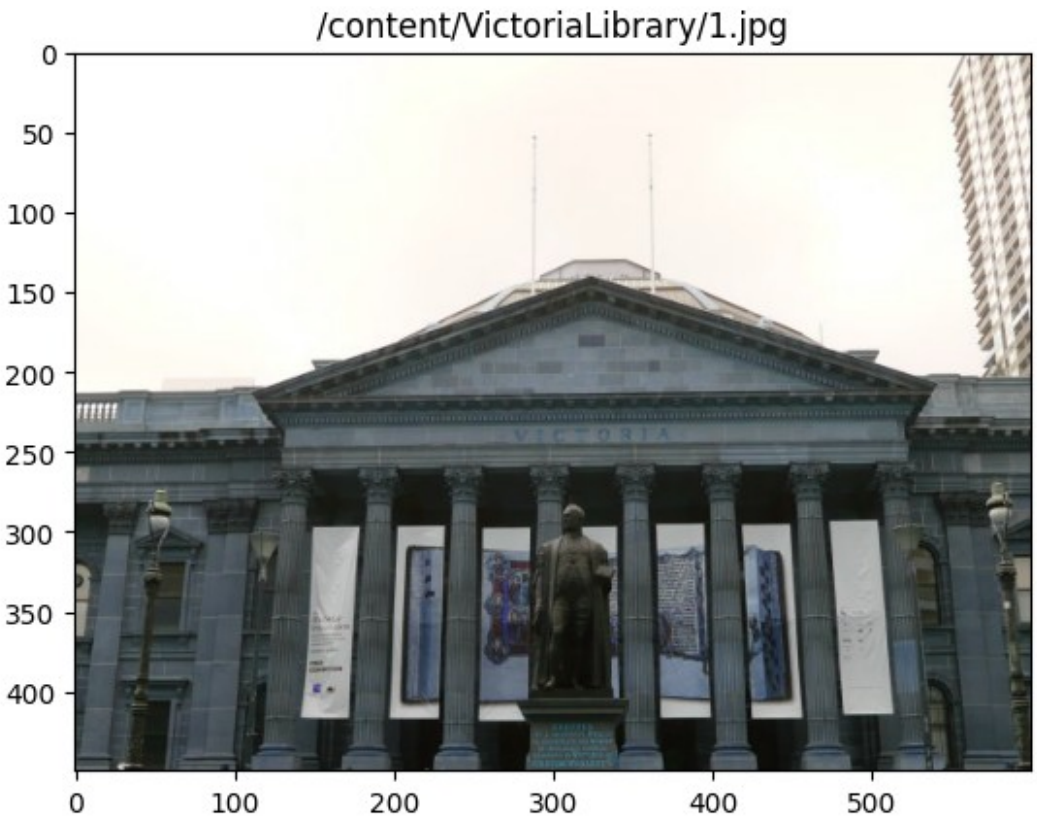
```
# Download training images from Google Drive
import gdown

# VictoriaLibrary
if test_each_cell:
  gdown.download_folder(id="14lVfW-ss6qo24PvcVdlBNW0o40u6QlOw",
quiet=True, use_cookies=False)

##########################################################################
########################
# The following dataset will be released later one week after the
release date of this project
# Until then, please work only on VictoriaLibrary Dataset
##########################################################################
########################

# Flatirons
# gdown.download_folder(id="1cwLOG_4keUaQHmp5RVwe_ZiAHeULpjOd",
quiet=True, use_cookies=False)

# CUBoulderSatView
# gdown.download_folder(id="10WcuZFuCxbe1FQSwpdy_9piyMM45eOI_",
quiet=True, use_cookies=False)

# CULogo
# gdown.download_folder(id="1Pv7eI-LOOUZxHffz1D5pmMKOk-Y7C-kx",
quiet=True, use_cookies=False)

# Checkerboard
# gdown.download_folder(id="1xPzK5se5AICqffjfuedqgiovQse4ILkS",
quiet=True, use_cookies=False)

# Indoors
# gdown.download_folder(id="1RfDSald_h2kwxRbcu8iq6ZPJ8TjOEGzK",
quiet=True, use_cookies=False)

# FlatironsChallenge
# gdown.download_folder(id="16_AV0IAfdr594jE_rK7serWNsVeN7wCS",
quiet=True, use_cookies=False)

# YourData
# gdown.download_folder(id="1xquJuwrwNBq0QEbwqB0LyJ6CAgHxa1lS",
quiet=True, use_cookies=False)

#####################################
## Plot Images using Matplotlib:
```

```
#######################################

# Import Libraries: (Feel free to modify according to your needs)
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy import ndimage
import numpy as np
import cv2
import os
from google.colab.patches import cv2_imshow

# Display Function for plotting an image using Matplotlib:
# You can also use CV2 plotting function
def show_img(img, title=""):
    plt.title(title)
    plt.imshow(img, cmap="gray")
    #plt.axis("off")
    plt.show()

if test_each_cell:
  jpg_files = []
  img_dir = "/content/VictoriaLibrary/"
  for root, dirs, files in os.walk(img_dir):
      for file in files:
          if file.endswith(".jpg"):
              jpg_files.append(os.path.join(root, file))
  print(jpg_files)

  img_array = []
  for f in jpg_files:
    img = cv2.imread(f)
    img_array.append(img)
    show_img(img, title=f)
```

```
['/content/VictoriaLibrary/2.jpg', '/content/VictoriaLibrary/3.jpg',
'/content/VictoriaLibrary/1.jpg']
```


/content/VictoriaLibrary/2.jpg
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


/content/VictoriaLibrary/3.jpg
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500



instead of hardcoding the files paths we can os.walk through a directory

Part B: Detect Corners [5 pts]

In this part, you will:

1. Detect corners
2. Display corners on top of the images

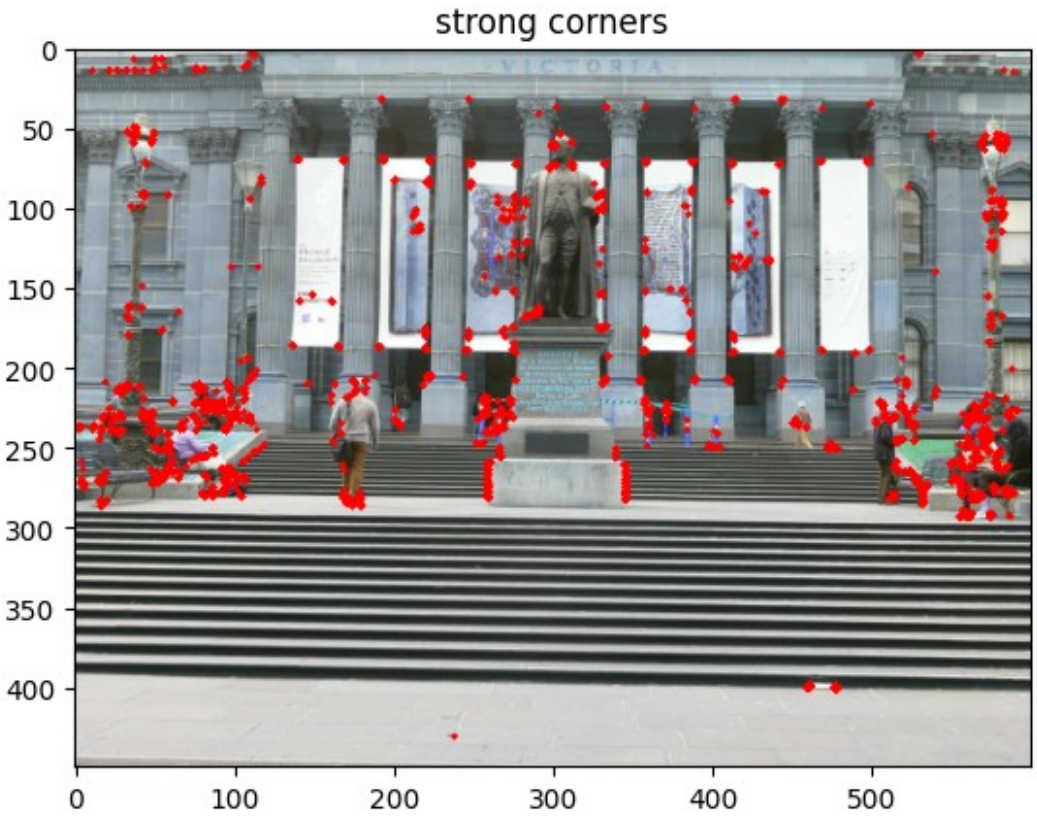
Sample Corner Detection:

The objective of this step is to detect corners such that they are equally distributed across the image in order to avoid weird artifacts in warping. Corners in the image can be detected using `cv2.cornerHarris` function with the appropriate parameters. The output is a matrix of corner scores: the higher the score, the higher the probability of that pixel being a corner.

```
# 1. Reduce the image size (optional) for faster processing
# 2. Convert Image to gray scale image (Try 'cv2.cvtColor(img,
cv2.COLOR_BGR2GRAY)')
# 3. Detect Corners (Use 'cv2.cornerHarris' or
'cv2.goodFeaturesToTrack' or equivalent functions)
# 4. Plot the corners on the image
# Refer:
```

```
https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html

# Note: Make sure you select and plot N_Strong corners i.e. Select the
corners with high corner response score
# Refer: https://medium.com/@erhan_arslan/delving-into-corner-
detection-with-python-harris-corner-detector-71231f9f44e2

# Refer:
# from scipy.ndimage import maximum_filter

def plot_corners(img, corners, title=""):
  img = img.copy()
  #img[corners[:, 0], corners[:, 1]] = [255, 0 ,0]
  for corner in corners:
    img = cv2.circle(img, (corner[1], corner[0]), 2, (255, 0, 0), -1)
  show_img(img, title=title)


'''
Detect Corners in the Image:
Input: Image
Output: N_Strong Corners and corner response map (cmap)
'''
def detect_corner(img, n = 2000, blockSize = 2, ksize = 3, k = 0.04,
resize_factor = 1, display_corners = False, display_title=""):
  # n: number of corners
  # blockSize: It is the size of neighbourhood considered for corner
detection
  # ksize: Aperture parameter of the Sobel derivative used
  # k: Harris detector free parameter in the equation.
  # resize_factor: resizes axes by multiplying by resize factor,
resize factor > 1 grows image, < 1 shrinks image
  if resize_factor != 1:
    img = img.copy()
    new_img_x = int(img.shape[1] * resize_factor)
    new_img_y = int(img.shape[0] * resize_factor)
    img = cv2.resize(img, (new_img_y, new_img_x))

  grey_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

  # Perform corner detection here:
  harris_output = cv2.cornerHarris(grey_img, blockSize, ksize, k)
  #print(type(harris_output))
  #print(harris_output.shape)
  #print(np.max(harris_output.flatten()))
  #print(np.min(harris_output.flatten()))

  flattened_output = harris_output.flatten()
  nth_most_confident = np.sort(flattened_output)[::-1][n]
  corner_y, corner_x = np.where(harris_output > nth_most_confident)
```

```
  #print(corner_x.shape)
  #print(corner_y.shape)
  corners = np.column_stack((corner_y, corner_x))
  #print(corners.shape)


  #thresholded_corners = np.zeros(harris_output.shape)
  #thresholded_corners[corners[:, 0], corners[:, 1]] = 1

  #print(thresholded_corners.shape)


  #show_img(harris_output, title="harris output")
  #show_img(thresholded_corners, title="thresholded_corners")
  # Return the a set of N_Strong Corners and corner response map
(cmap)
  if display_corners:
    plot_corners(img, corners, title=display_title)
  return corners, harris_output

if test_each_cell:
  strong_corners, cmap = detect_corner(img_array[0], display_corners =
True, display_title="strong corners")
  print(strong_corners.shape)
  print(cmap.shape)
```




```
(2000, 2)
(450, 600)
```

Part 3: ANMS [10 pts]

In this part, you will:

1. Select N best corners that are equally spaced through out the image using ANMS (Adaptive Non-Maximal Suppression)
2. Create Feature Descriptors from Corners

Sample Corner Detection after ANMS:

To find particular strong corners that are spread across the image, first we need to find N_{strong} corners. Because, when you take a real image, the corner is never perfectly sharp, each corner might get a lot of hits out of the N_{strong} corners - we want to choose only the N_{best} corners after ANMS. In essence, you will get a lot more corners than you should! ANMS will try to find corners which are local maxima.

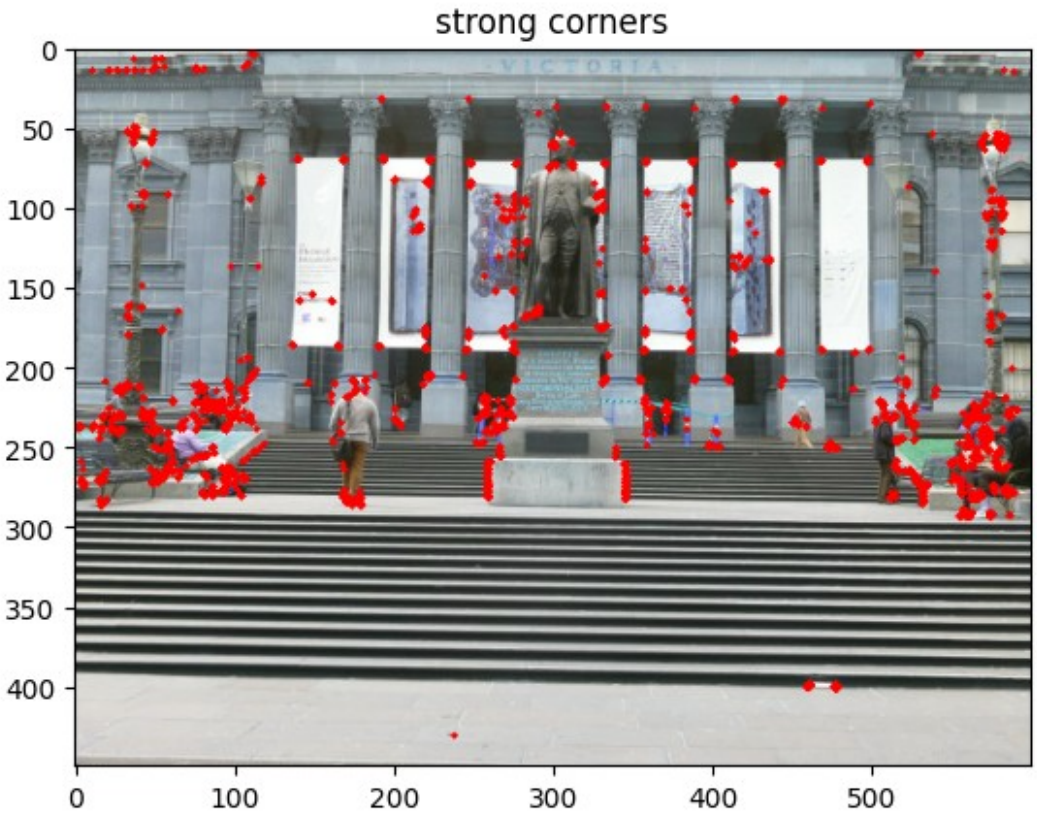
ANMS Algorithm:

Note: Python's `scipy.ndimage.filters.maximum_filter` is similar to `imregionalmax` from MATLAB.

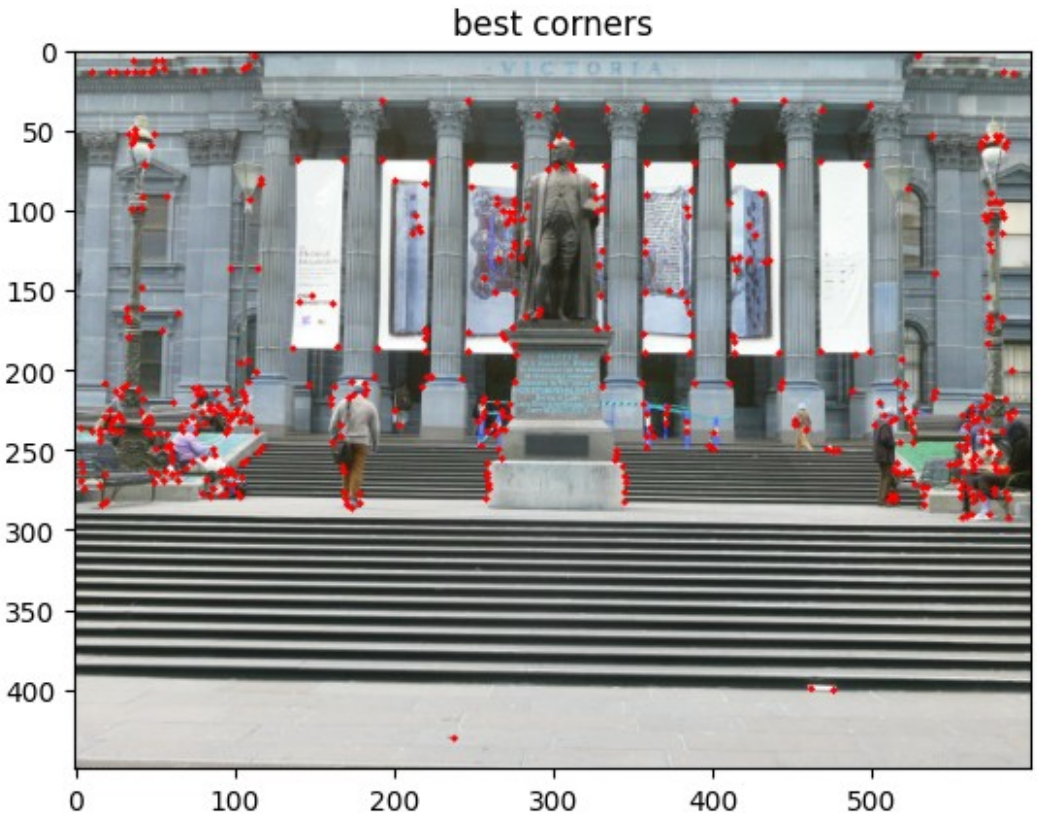
Intuitively, we are trying to find N_{best} corners, say `200` from `500` N_{strong} corners such that they are equally distant from one another. To do sp, we define an adaptive radius r_i as the distance between two feature points `i` and `j`.

```
'''
ANMS Algorithm:
Input: Corner Response Map, N_Strong Corner Set, nBest Number of Best
Corners
Output: N_Best Corners Set
'''
def anms(cmap, N_Strong, nBest, display_corners = False,
display_title="", img=None):
  # Perform ANMS here:
  # Return the a set of N_best Corners
  R = np.full(N_Strong.shape[0], np.inf)
  #print(R)
  #print(R.shape)


  for i in range(N_Strong.shape[0]):
    for j in range(N_Strong.shape[0]):
      ED = np.inf
      activation_i = cmap[N_Strong[i, 0], N_Strong[i, 1]]
      activation_j = cmap[N_Strong[j, 0], N_Strong[j, 1]]
      if activation_i > activation_j:
        ED = np.sum((N_Strong[i] - N_Strong[j])**2)
      #print(f"ED == 1 {i}, {j}") if ED == 1 else None
      if ED < R[i] and i != j:
        R[i] = ED

  #print(R)
  #print(R.shape)
  #print(N_Strong.shape)
  r_corner_zip = zip(R, N_Strong)
  r_corner_zip = sorted(r_corner_zip, key=lambda x: x[0], reverse =
True)
  #r_corner_zip = sorted(r_corner_zip, key=lambda x: x[0], reverse =
False)
  top_n_r_corners_zip = r_corner_zip[:nBest]
  #for ri, corner in top_n_r_corners_zip:
  #  print(f"ri: {ri}, corner:{corner}")
```

```
  top_n_corners = [corner for ri, corner in top_n_r_corners_zip]
  top_n_corners = np.array(top_n_corners)
  if display_corners:
    plot_corners(img, top_n_corners, title=display_title)
  return top_n_corners

if test_each_cell:
  img_index = 0
  strong_corners, cmap = detect_corner(img_array[img_index], n=2000,
display_corners = True, display_title = "strong corners")
  print(f"N_strong: {strong_corners.shape}")
  best_corners = anms(cmap, strong_corners, 500, display_corners =
True, display_title = "best corners", img=img_array[img_index])
  print(f"N_best: {best_corners.shape}")
```




```
N_strong: (2000, 2)
```




```
N_best: (500, 2)
```

Part 4: Feature Descriptors [15 pts]

In the previous step, you found the feature points (locations of the N best best corners after ANMS are called the feature points). You need to describe each feature point by a feature vector, this is like encoding the information at each feature points by a vector. One of the easiest feature descriptor is described next. Note: The following kernel/patch/vector sizes are user chosen. Feel free to modify them according to your data

---

Sample Feature Desciptors:

---

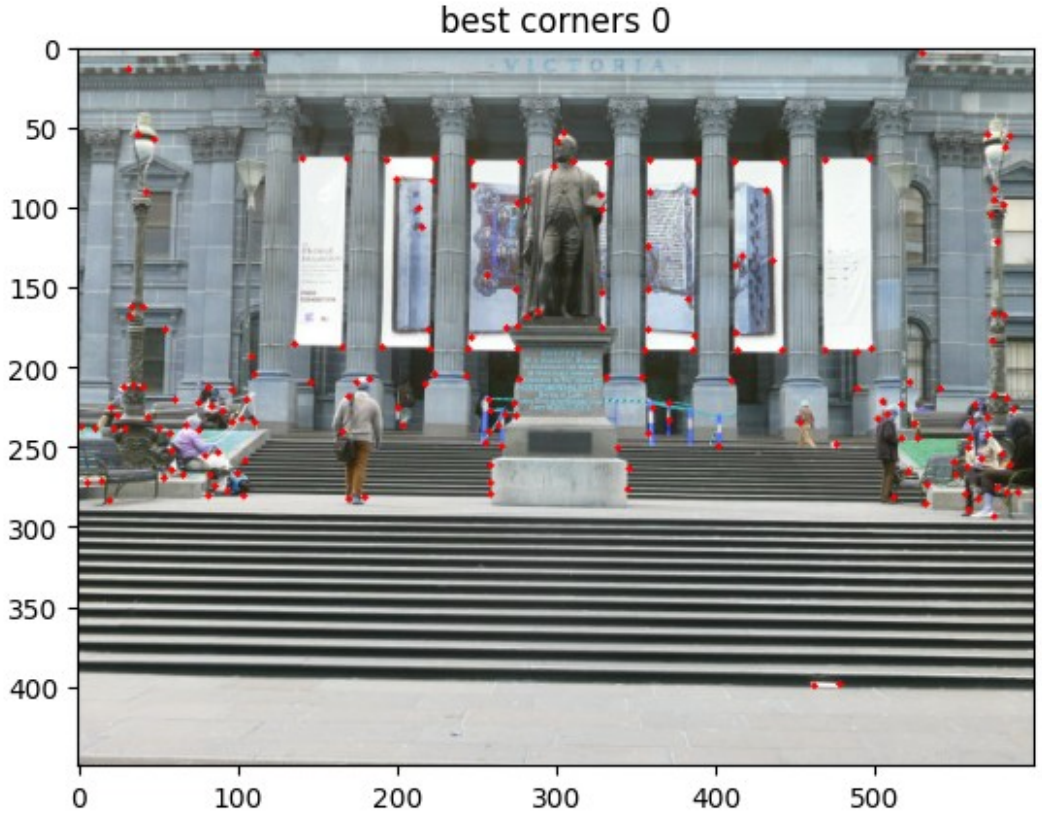
1. Take a patch of size `40×40` centered (this is very important and depends on the input image resolution) around the keypoint.
2. Now apply `gaussian blur` (feel free to play around with the parameters and use `cv.GaussianBlur()` function).
3. Now, sub-sample the blurred output (this reduces the dimension) to `8×8`.

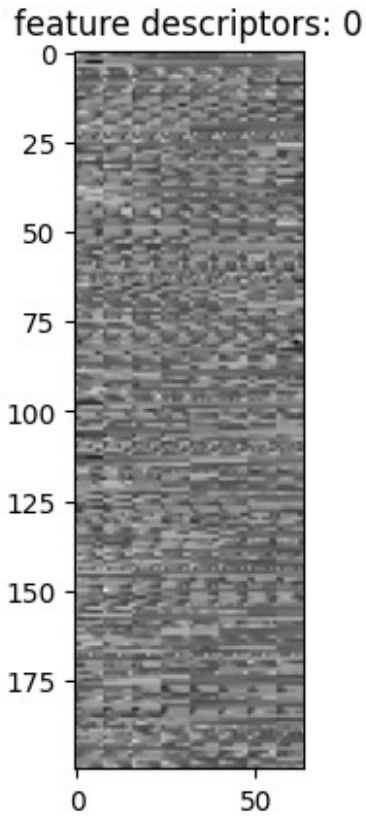
4. `reshape` it to obtain a `64×1` vector.
5. `Standardize` the vector to have `mean` of `0` and `variance` of `1` (This can be done by subtracting all values by mean and then dividing by the standard deviation). Note: Standardization is used to remove bias and some illumination effect.

```
'''
Feature Descriptor:
Input: Corner Coordinates, Image
Output: Feature Descriptors
'''
def feature_desc(gray_img, N_Best, patch_side_length = 40, Blur_Kernel
= (3,3), subsample_side_length = 16, blur_std = 0, display_descriptors
= False, display_title=""):
  # Generate Feature Descriptor For Corners Here
  # 1. Define Patch Size, Blur Kernels and Subsample Size
  # 2. Extract the Patch Centered Around Corners (Ensure the Patch is
within the image boundaries)
  # 2. Apply Gaussian Blur and Subsample
  # 3. Reshape Subsampled patch to a 64x1 vector and standardize the
descriptor
  # Return Descriptors and their Coordinates

  #print(gray_img.shape)
  #show_img(gray_img, title="original_gray_img")
  offset = patch_side_length // 2
  img_height = gray_img.shape[0]
  img_width = gray_img.shape[1]
  #print(img_height)
  #print(img_width)
  #gray_img = gray_img.copy()
  #gray_img = cv2.copyMakeBorder(gray_img, offset, offset, offset,
offset, cv2.BORDER_CONSTANT, value = 0)
  #cropped_gray_img = gray_img[offset:original_img_height-offset,
offset:original_img_width-offset]
  #show_img(cropped_gray_img, title="cropped_gray_img")

  #print(gray_img.shape)
  #show_img(gray_img, title="gray_img")
  #print(N_Best)

  #print()
  N_Best = N_Best.copy()
  N_Best += offset
  #print(gray_img.shape)
  #print(N_Best)
  desc = []
  #acc = 0
  for corner in N_Best:
```

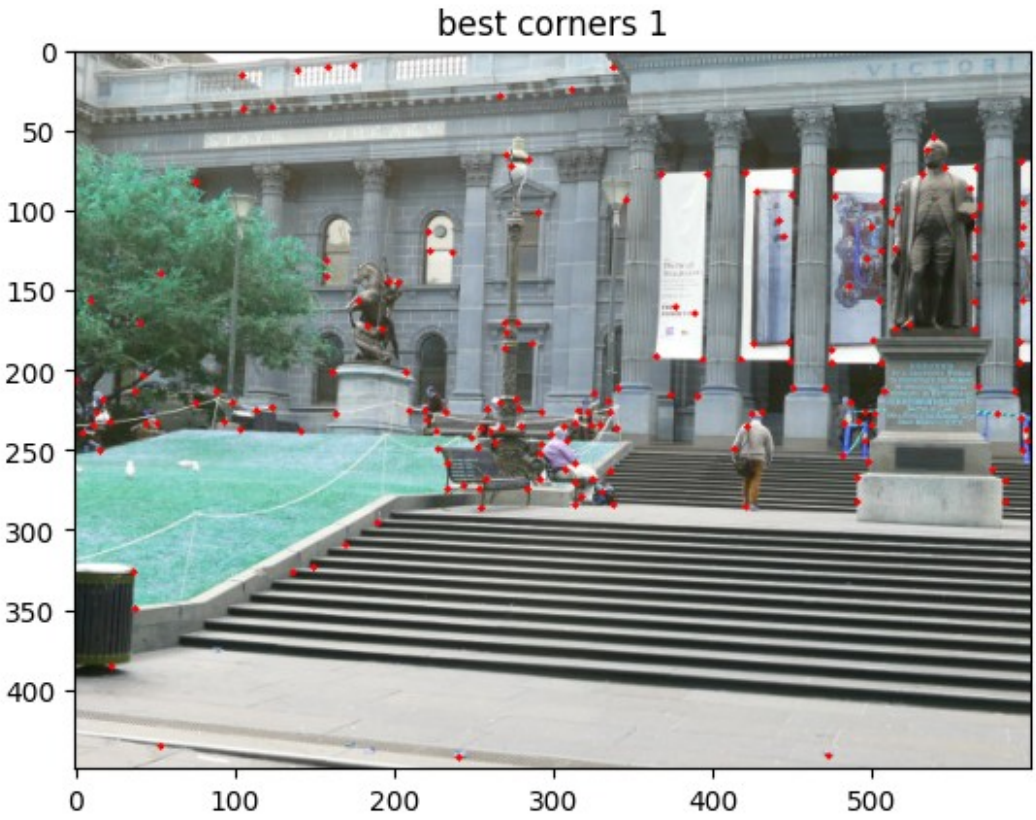
```
    lower_left_x = corner[1] - offset
    lower_left_y = corner[0] - offset

    upper_right_x = corner[1] + offset
    upper_right_y = corner[0] + offset

    if lower_left_x < 0 or lower_left_y < 0:
      pass
      #raise ValueError(f"Corner ({corner}) patch lower_left_x
({lower_left_x}) or lower_left_y ({lower_left_y}) is out of bounds for
shape: {gray_img.shape}")
    if upper_right_x > gray_img.shape[1] or upper_right_y >
gray_img.shape[0]:
      pass
      #raise ValueError(f"Corner ({corner}) patch
upper_right_x({upper_right_x}) or upper_right_y({upper_right_y}) is
out of bounds for shape: {gray_img.shape}")


    #gray_img[corner[0], corner[1]] = 255
    patch = gray_img[lower_left_y:upper_right_y,
lower_left_x:upper_right_x]

    blured_patch = cv2.GaussianBlur(src=patch, ksize=Blur_Kernel,
sigmaX=blur_std)

    pooled_image = cv2.resize(blured_patch, (subsample_side_length,
subsample_side_length), interpolation=cv2.INTER_AREA)
    feature_vector = pooled_image.flatten().astype(np.float64)
    feature_vector -= np.mean(feature_vector)
    std = np.std(feature_vector)
    if std != 0:
      feature_vector /= std
    else:
      feature_vector = np.zeros_like(feature_vector)
    desc.append(feature_vector)

    #show_img(patch, title=f"patch {acc}")

#show_img(feature_vector.reshape(subsample_side_length,subsample_side_
length), title=f"feature vector {acc}")
    #acc+=1


  desc = np.array(desc)
  #np.random.shuffle(desc)
  if display_descriptors:
    show_img(desc, title=display_title)
  return desc, N_Best
```

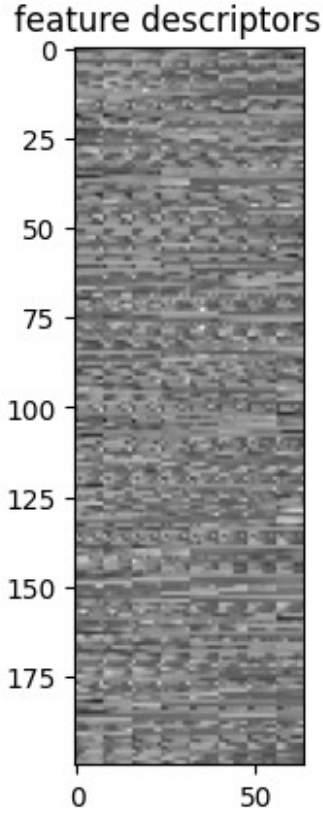
```
#grey_img = cv2.cvtColor(img_array[img_index], cv2.COLOR_BGR2GRAY)
#desc, des_coords = feature_desc(grey_img, best_corners,
display_descriptors=True, display_title = f"feature descriptors:
{img_index}")
if test_each_cell:
  for img_index in range(len(img_array)):
    strong_corners, cmap = detect_corner(img_array[img_index], n=1000,
display_corners = False)
    best_corners = anms(cmap, strong_corners, 200, display_corners =
True, display_title = f"best corners {img_index}",
img=img_array[img_index])
    grey_img = cv2.cvtColor(img_array[img_index], cv2.COLOR_BGR2GRAY)
    desc, des_coords = feature_desc(grey_img, best_corners,
display_descriptors=True, display_title = f"feature descriptors:
{img_index}", subsample_side_length=8)
    print(f"descriptors: {desc.shape}")
    print(f"descriptor coords: {des_coords.shape}")
```



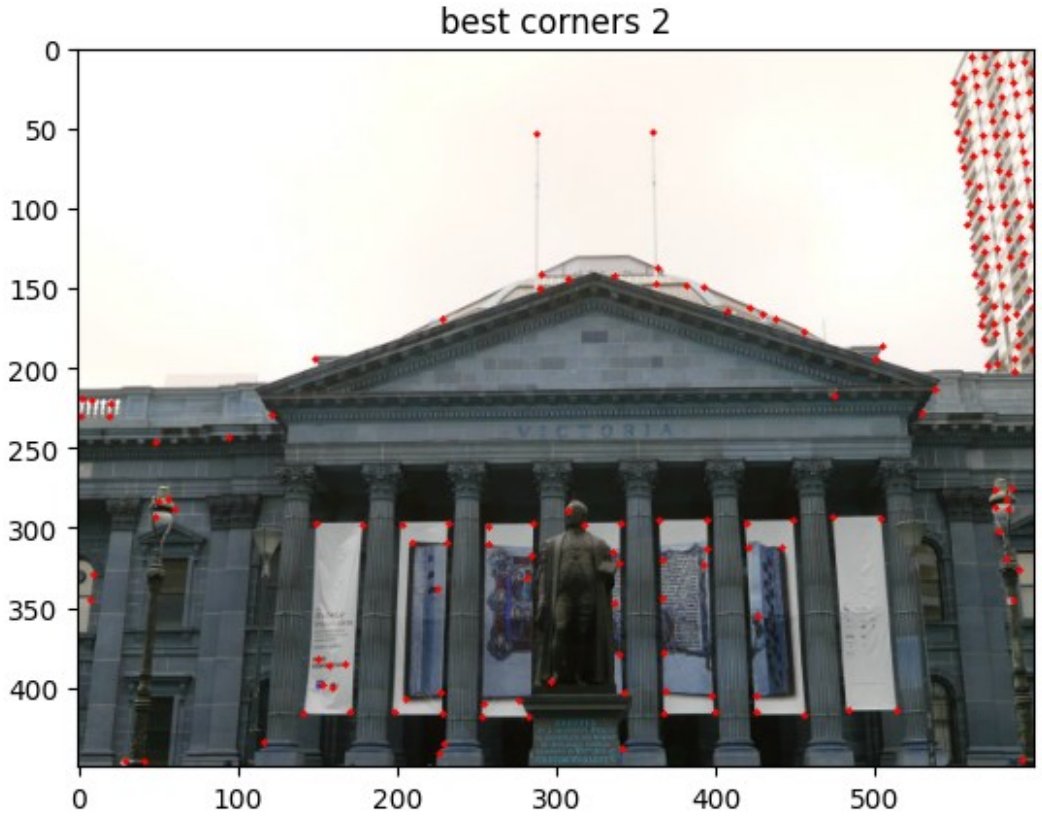


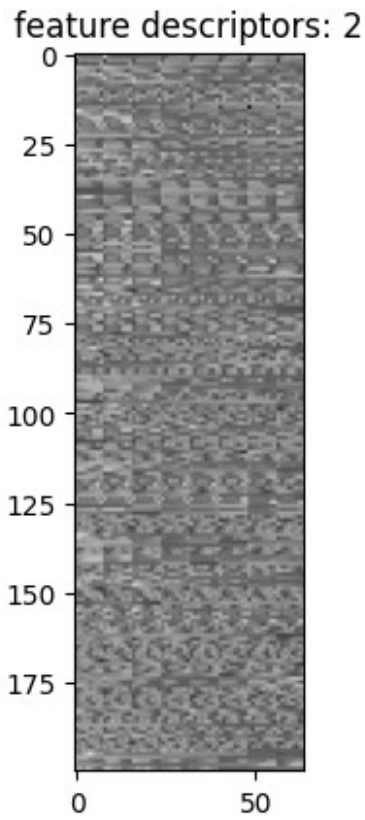

```
descriptors: (200, 64)
descriptor coords: (200, 2)
```


best corners 1
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


feature descriptors: 1
0
25
50
75
100
125
150
175
0
50

```
descriptors: (200, 64)
descriptor coords: (200, 2)
```






```
descriptors: (200, 64)
descriptor coords: (200, 2)
```

Part 5: Feature Matching [15 pts]

In the previous step, you encoded each keypoint by `64×1` feature vector. Now, you want to match the feature points among the two images you want to stitch together. In computer vision terms, this step is called as finding feature correspondences or feature matching between the 2 images.

Sample Feature Matching: (Note some errors and bad matches)

---

1. Pick a point in image 1, compute sum of square difference between all points in image 2.
2. Take the ratio of best match (lowest distance) to the second best match (second lowest distance) and if this is below some ratio keep the matched pair or reject it. Repeat this for all points in image 1.

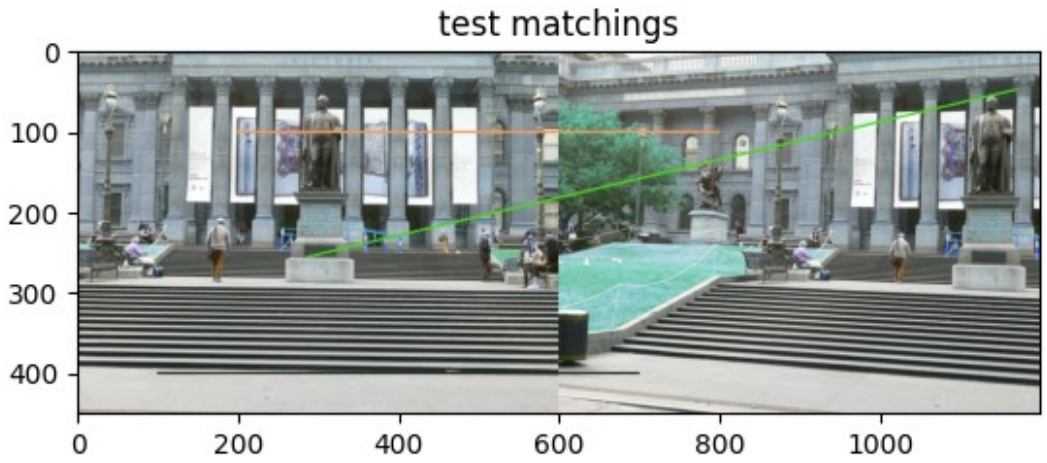
$$SSD = \sum (v_1 - v_2)^2$$

3. You will be left with only the confident feature correspondences and these points will be used later to estimate the transformation between the 2 images also called as Homography.

Brute-force matching compares every descriptor in one image to every descriptor in another image. While this method guarantees finding matches, it's computationally expensive and may not produce the best results due to its non-selective nature.

```
def plot_corrispondances(img1, img2, matches, title=""):
  img1 = img1.copy()
  img2 = img2.copy()
  matches = matches.copy()

  #show_img(img1, title=f"x:{img1.shape[1]} y:{img1.shape[0]}")
  #show_img(img2, title=f"x:{img2.shape[1]} y:{img2.shape[0]}")

  new_window_x = img1.shape[1] + img2.shape[1]
  new_window_y = max(img1.shape[0], img2.shape[0])
  img_window = np.zeros((new_window_y, new_window_x, 3),
dtype=np.uint8)
  #show_img(img_window, title=f"x:{img_window.shape[1]} y:
{img_window.shape[0]}")

  #print(f"img_window.shape: {img_window.shape}")



#show_img(img_window[:img_array[0].shape[0], :img_array[0].shape[1]],
title="first image section")
  img_window[:img1.shape[0], :img1.shape[1]] = img1




  #print(img_array[0].shape[0])
  #print(img_array[1].shape[1])
  #show_img(img_window[:img_array[0].shape[0],
img_array[0].shape[1]:], title="second image section")
  img_window[:img2.shape[0], img1.shape[1]:] = img2

  #cv2.circle(img_window, (0+img1.shape[1], 0), 50, (255, 0, 0), -1)
  #show_img(img_window, title="tmp")


  for m in matches:
    img_1_coords = m[:2]
    img_2_coords = m[2:]
    #print(f"img_1_coords:  {img_1_coords}")
```

```
    #print(f"img_2_coords:  {img_2_coords}")
    img_2_coords[1] += img1.shape[1]
    #print(f"adjusted img_2_coords:  {img_2_coords}")
    #print()
    random_color = tuple(np.random.randint(0, 255, 3).tolist())
    cv2.line(img_window, img_1_coords[::-1], img_2_coords[::-1],
random_color, 2)

  show_img(img_window, title=title)

if test_each_cell:
  matches = np.array([
      [400, 100, 400, 100],
      [100, 200, 100, 200],
      [255, 288, 47, 571]
  ])
  plot_corrispondances(img_array[0], img_array[1], matches.copy(),
title="test matchings")
```

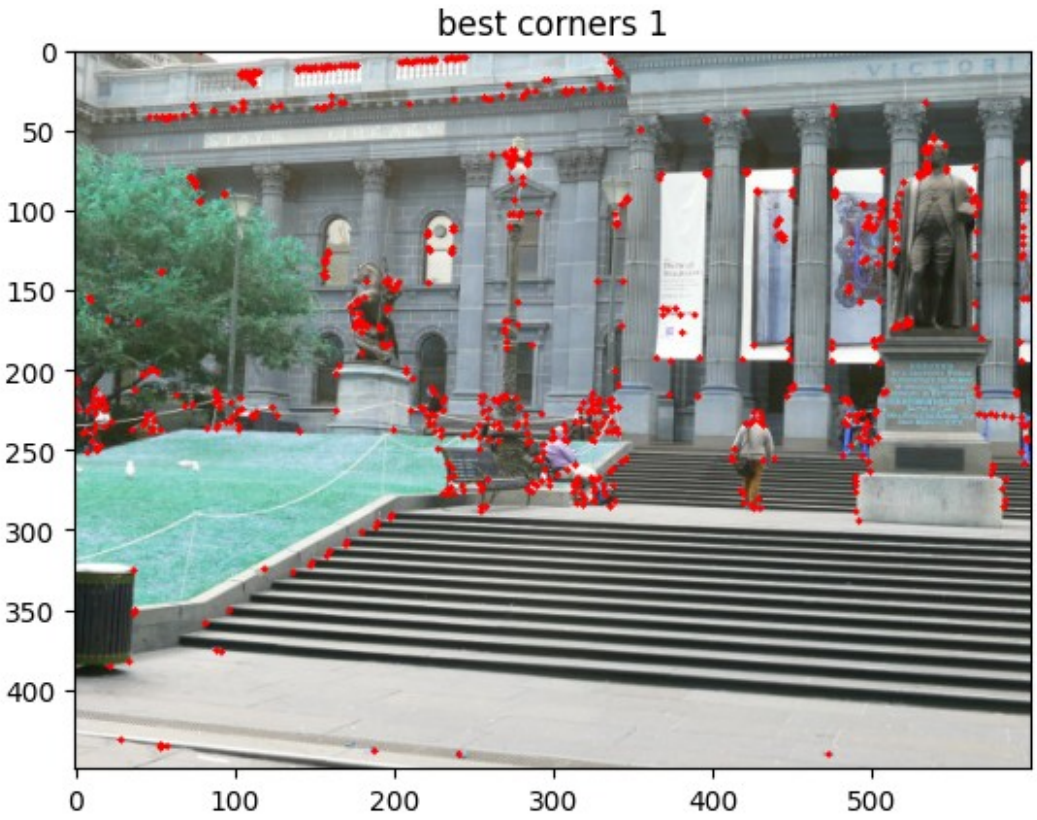


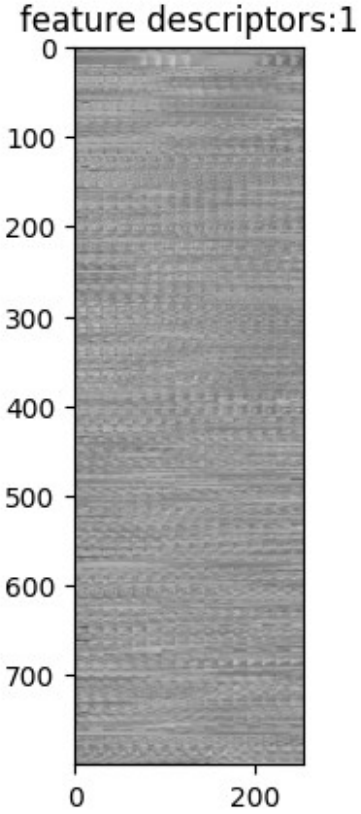

```
'''
Feature Matching
'''

def feature_matching(desc1, desc2, desc1_coords, desc2_coords,
img1=None, img2=None, ratio_threshold = 0.9, do_print = False,
display=True):
  # Perform Feature Matching
  # Calculate the Sum Square Difference between two different feature
descriptors (SSD)
  # Apply ratio test - check if the distance to the best match is
significantly smaller than the distance to the second-best match.
      # If the ratio test passes, add the match to the list of
matches.
      # Store the corresponding keypoints from image 1.
```

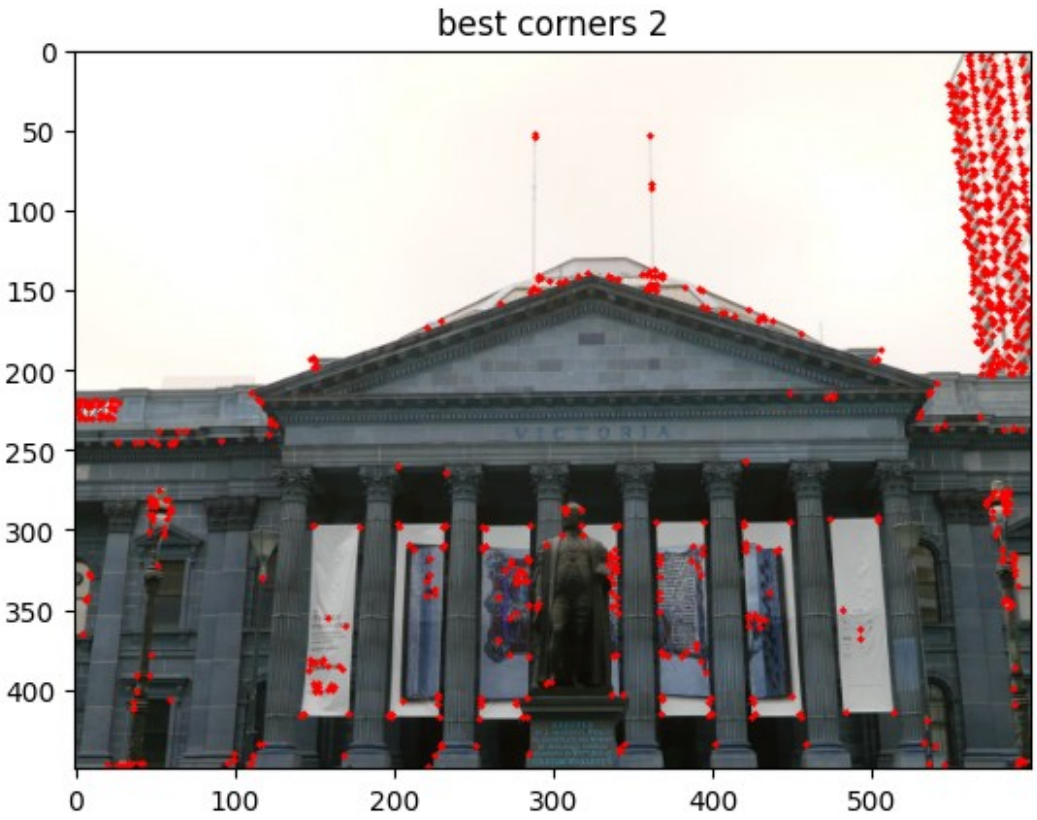
```
      # Store the corresponding keypoints from image 2.
  # Return Feature Matches
  candidate_matches = {}
  for i in range(desc1.shape[0]):

    v1 = desc1[i]


    differences = desc2 - v1


    square_differences = differences**2


    sum_square_differences = np.sum(square_differences, axis=1)


    sorted_distances_indicies = np.argsort(sum_square_differences)


    best_match_distance_index = sorted_distances_indicies[0]

    second_best_match_distance_index = sorted_distances_indicies[1]


    best_match_distance =
sum_square_differences[best_match_distance_index]

    second_best_match_distance =
sum_square_differences[second_best_match_distance_index]


    #lower is better so we want (small)/(big) which is close to 0
    distance_ratio = best_match_distance / second_best_match_distance

    if do_print:
      print(f"v1: {v1}")
      print(v1.shape)
      print()

      print(f"desc2: {desc2}")
      print(desc2.shape)

      print(f"differences: {differences}")
      print(differences.shape)
      print()

      print(f"square_differences: {square_differences}")
      print(square_differences.shape)
      print()
```

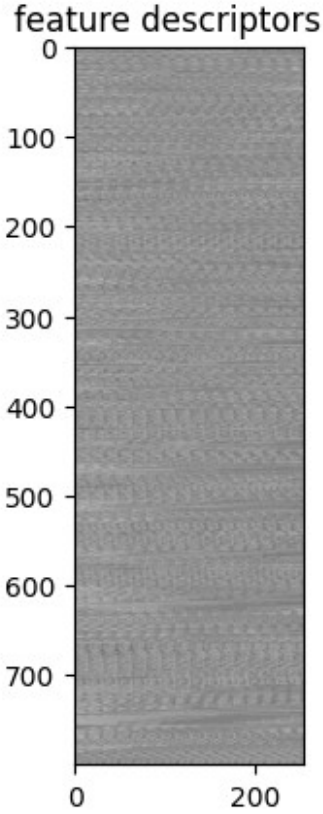
```
      print(f"sum_square_differences: {sum_square_differences}")
      print(sum_square_differences.shape)
      print()

      print(f"sorted_distances_indicies: {sorted_distances_indicies}")
      print(sorted_distances_indicies.shape)
      print()

      print(f"best_match_distance_index: {best_match_distance_index}")
      print(f"second_best_match_distance_index:
{second_best_match_distance_index}")

      print(f"best_match_distance: {best_match_distance}")
      print(f"second_best_match_distance:
{second_best_match_distance}")
      print(f"distance_ratio: {distance_ratio}")

    #numerator is smaller than denominator -> 0
    #everything < threshhold should be accepted
    if distance_ratio < ratio_threshold:
      print(f"appending match: {desc1_coords[i]}-
>{desc2_coords[best_match_distance_index]}") if do_print else None
      m = (np.concatenate((desc1_coords[i],
desc2_coords[best_match_distance_index])), best_match_distance)
      if best_match_distance_index in candidate_matches:
        candidate_matches[best_match_distance_index].append(m)
      else:
        candidate_matches[best_match_distance_index] = [m]
  #print(f"candidate_matches: {candidate_matches}")
  matches = []
  for desc2_match_index, matches_list in candidate_matches.items():
      best_match = min(matches_list, key=lambda x: x[1])
      #print(best_match)
      #print(best_match[0])
      matches.append(best_match[0])
  matches = np.array(matches)
  if img1 is not None and img2 is not None and display:
    plot_corrispondances(img1, img2, matches, title="all feature
matching")


  return matches

if test_each_cell:
  num_strong_corners = 2000
  num_best_corners = 800

  strong_corners_1, _ = detect_corner(img_array[1],
n=num_strong_corners)
  best_corners_1 = anms(cmap, strong_corners_1, num_best_corners,
```

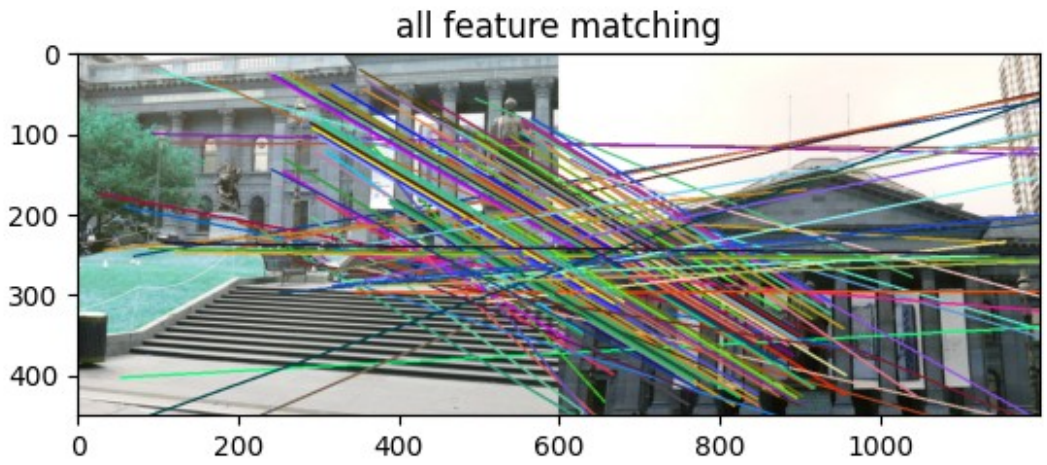
```
display_corners = True, display_title = f"best corners 1",
img=img_array[1])
  grey_img_1 = cv2.cvtColor(img_array[1], cv2.COLOR_BGR2GRAY)
  desc_1, des_coord_1 = feature_desc(grey_img_1, best_corners_1,
display_descriptors=True, display_title = f"feature descriptors:1")

  strong_corners_2, _ = detect_corner(img_array[2],
n=num_strong_corners)
  best_corners_2 = anms(cmap, strong_corners_2, num_best_corners,
display_corners = True, display_title = f"best corners 2",
img=img_array[2])
  grey_img_2 = cv2.cvtColor(img_array[2], cv2.COLOR_BGR2GRAY)
  desc_2, des_coord_2 = feature_desc(grey_img_2, best_corners_2,
display_descriptors=True, display_title = f"feature descriptors: 1")



  matches = feature_matching(desc_1, desc_2, des_coord_1, des_coord_2,
img1 = img_array[1], img2 = img_array[2])
  n = matches.shape[0]
  k = 1
  for i in np.linspace(0,1,num=k):
    start = int(i*n)
    end = int((i + 1/k)*n)
    matches_subset = matches
    plot_corrispondances(img_array[1], img_array[2],
matches[start:end], title=f"feature matching {start}:{end}")
  print(matches.shape)
  #print(matches)
```

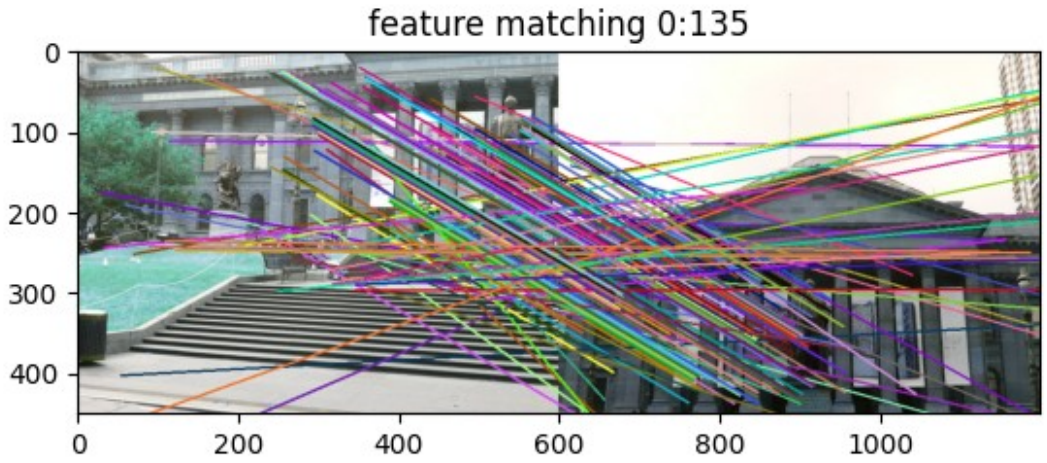

best corners 1
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


feature descriptors:1
0
100
200
300
400
500
600
700
0
200


best corners 2
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


feature descriptors: 1
0
100
200
300
400
500
600
700
0
200






```
(135, 4)
```

Part 6: Refine Matches [15 pts]

We now have matched all the features correspondences but not all matches will be right. To remove incorrect matches, we will use a robust method called Random Sampling Concensus or RANSAC to compute homography.

RANSAC, or Random Sample Consensus, is an iterative algorithm used to estimate the parameters of a mathematical model. In the context of image stitching, we can use RANSAC to estimate the transformation that maps points from one image to another. (Refer: Wiki)

Sample Refined Feature Matching: (Note the bad feature matches disappeared)

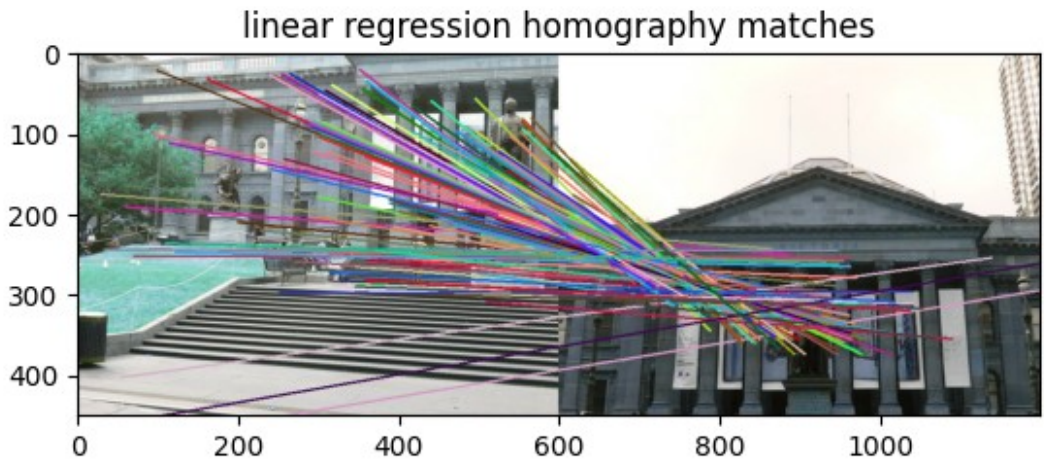
---

1. Select four feature pairs (at random), p_i from `image 1`, p_i^1 from `image 2`.
2. Compute homography H (exact). Write a function `est_homography`.
3. Compute inliers where $SSD\left(p_i^1, H\,p_i\right)$<thresh. Here, $H\,p_i$ computed using another function that you will be writing `apply_homography`.
4. Repeat the last three steps until you have exhausted $N_{\max}$ number of iterations (specified by user) or you found more than percentage of inliers (Say 90% for example).
5. Keep the largest set of inliers
6. Re-compute the least-square $\widehat{H}$ estimate on all of the inliers.

```
def est_homography(matches):
  ones = np.ones((matches.shape[0], 1))
  X = matches[:, :2]
  X = np.hstack((X, ones))

  Y = matches[:, 2:]
  Y = np.hstack((Y,ones))
  #xcoord_labels = Y[:, 0]
  #ycoord_labels = Y[:, 1]

  #print(f"X.shape:{X.shape}")
  #print(X)
  #print(f"Y.shape:{Y.shape}")
  #print(Y)

  #print(f"xcoord_labels.shape:{xcoord_labels.shape}")
  #print(xcoord_labels)
  #print(f"ycoord_labels.shape:{ycoord_labels.shape}")
  #print(ycoord_labels)

  #x_coord_linear_regression = np.linalg.inv(X.T @ X) @ X.T @
xcoord_labels
  #y_coord_linear_regression = np.linalg.inv(X.T @ X) @ X.T @
ycoord_labels
  try:
    linear_regression_solution = np.linalg.inv(X.T @ X) @ X.T @ Y
  except np.linalg.LinAlgError:
    linear_regression_solution = np.linalg.pinv(X.T @ X) @ X.T @ Y


  #print(f"x_coord_linear_regression:
{x_coord_linear_regression.shape}\n {x_coord_linear_regression}")
  #print(f"y_coord_linear_regression:
{y_coord_linear_regression.shape}\n {y_coord_linear_regression}")
  #print(f"linear_regression_solution:
{linear_regression_solution.shape}\n {linear_regression_solution}")


  #homography = np.vstack([x_coord_linear_regression,
y_coord_linear_regression])
```

```
  #print(f"homography:{homography.shape}\n {homography}")
  determinant = np.linalg.det(linear_regression_solution)
  is_singular = np.isclose(determinant, 0)
  if is_singular:
    epsilon = 1e-10  # Very small noise
    linear_regression_solution = linear_regression_solution +
np.random.normal(0, epsilon, linear_regression_solution.shape)

  return linear_regression_solution

def apply_homography(homography_matrix, matches, img1=None, img2=None,
title=""):
  ones = np.ones((matches.shape[0], 1))
  X = matches[:, :2]
  X = np.hstack((X, ones))

  #print(f"X.shape:{X.shape}")

  predicted = X @ homography_matrix
  #print(f"predicted.shape:{predicted.shape}")
  if img1 is not None and img2 is not None:
    X = X[:, :2].astype(np.int32)
    p = predicted[:, :2].astype(np.int32)
    #print(f"X:{X.shape}")
    #print(X)
    #print(f"p:{p.shape}")
    #print(predicted)
    expected_matches = np.hstack((X, p))
    #print(f"expected_matches:{expected_matches.shape}")
    #print(expected_matches)


    print(expected_matches.shape)
    plot_corrispondances(img1, img2, expected_matches, title=title)
  return predicted
if test_each_cell:
  homo = est_homography(matches)
  print(f"homography: {homo.shape}\n{homo}")
  transformed_matches = apply_homography(homo, matches[:, :2], img1 =
img_array[1], img2=img_array[2], title="linear regression homography
matches")
```

```
homography: (3, 3)
[[ 2.60621356e-02  1.18717197e+00 -1.81603864e-18]
 [ 2.25606623e-01  1.37936278e-01  2.41234983e-18]
 [ 2.31936727e+02  5.42546866e+01  1.00000000e+00]]
(135, 4)
```

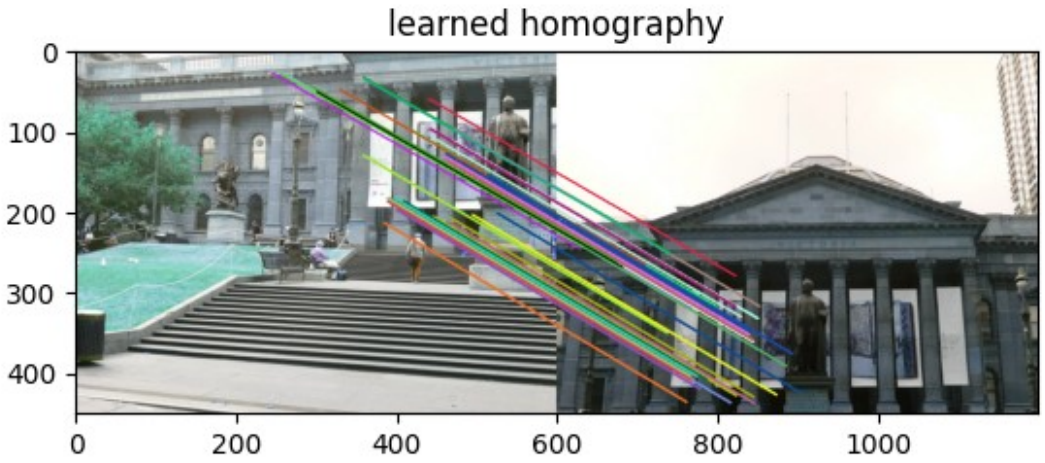



```
'''
Performs RANSAC to find the best homography matrix.
  Input:
    matches: an nx4 matrix where columns are x1, y1 x2, y2 where the
corner in image 1 x1,y1 matches to a corner in image 2 x2 y2
    iter: The maximum number of iterations.
    N: The number of inliers required for a model to be considered
good.
    thr: The distance threshold for a point to be considered an
inlier.

  Returns:
    H: The best homography matrix found.
    inliers: an nx4 matrix where columns are x1, y1 x2, y2 where the
corner in image 1 x1,y1 matches to a corner in image 2 x2 y2
'''
def RANSAC(matches, iter, N, thr, sample_size = 4, img1 = None, img2 =
None, display=True):
  # Perform RANSAC
  # Return Homography
  best_matrix = None
  best_N_inliers = -np.inf
  best_inliers = None
  best_mean_error = np.inf

  ones = np.ones((matches.shape[0], 1))
  true_positions = matches[:, 2:]
  true_positions = np.hstack((true_positions, ones))

  for i in range(iter):
    random_row_indicies = np.random.choice(matches.shape[0],
sample_size, replace=False)
    #print(f"random_row_indicies: {random_row_indicies}")
    #print(random_row_indicies.shape)
    #print()
```

```
    random_matches = matches[random_row_indicies, :]
    #print(f"random_matches:{random_matches}")
    #print(random_matches.shape)
    #print()

    homograph_matrix = est_homography(random_matches)
    predicted_positions = apply_homography(homograph_matrix,
matches[:, :2])

    #print(f"predicted_positions: {predicted_positions.shape}")
    #print(f"true_positions: {true_positions.shape}")
    square_prediction_differences = (predicted_positions -
true_positions)**2
    sum_square_prediction_differences =
np.sum(square_prediction_differences, axis=1)
    sum_prediction_error = np.sqrt(sum_square_prediction_differences)
    #print(f"sum_square_prediction_differences:
{sum_square_prediction_differences.shape}")
    #print(f"sum_square_prediction_differences:
{sum_square_prediction_differences}")
    mean_error = np.mean(sum_prediction_error)
    mask = sum_prediction_error < thr
    #print(f"mask: {mask.shape}")
    #print(f"mask: {mask}")
    inliers = matches[mask, :]
    #print(f"inliers: {inliers.shape}")

    if inliers.shape[0] > N:
      if img1 is not None and img2 is not None and display:
        print(f"returning matrix because we have > {N} inliers, mean
error: {mean_error}")
        apply_homography(homograph_matrix, inliers, img1=img1,
img2=img2, title="learned homography")
      return homograph_matrix, inliers

    if inliers.shape[0] > best_N_inliers:
      best_N_inliers = inliers.shape[0]
      best_matrix = homograph_matrix
      best_inliers = inliers
      best_mean_error = mean_error




  if img1 is not None and img2 is not None and display:
    print(f"returning matrix with {best_N_inliers} inliers and mean
error {best_mean_error}, ranout of iterations")
    apply_homography(best_matrix, best_inliers, img1=img1, img2=img2,
title = "learned homography")
  return best_matrix, best_inliers
```

```
if test_each_cell:
  # Plot your updated feature matching results here (and visually
compare them with the previous steps)
  homo, inliers = RANSAC(matches, iter = 1000, N = 0.9 *
matches.shape[0], thr = 2, sample_size = 4, img1 = img_array[1],
img2=img_array[2])
  #print(homo.shape)
  #print(inliers.shape)
```

```
returning matrix with 36 inliers and mean error 162.52069604890065,
ranout of iterations
(36, 4)
```



Part 7: Warp and Blend [15 pts]

Note: 15 pts extra credit for implementing everything from scratch for this section.

Panorama can be produced by overlaying the pairwise aligned images to create the final output image. Understanding `cv2.warpPerspective` and `cv2.perspectiveTransform` functions will be helpful. For such implementation, apply bilinear tranpolation when you copy pixel values. Feel free to use any third party code for warping and transforming images.

Sample Final Warping and Blending Results: (Note the bad feature matches disappeared)

For a few datasets, you might need to perform cylindrical projection before warping and blending images together. This is because a simple projective transform such as homography will produce substandard results and the images will be stretched/shrunken to a large extent over the edges. Note: Feel free to use any open source code for this part.

To overcome such distortion problem at the edges, we will be using cylinderical projection on the images first before performing other operations. Essentially, this is a pre-processing step. The following equations transform between normal image co-ordinates and cylinderical co-ordinates:

$$ x' = f \cdot tan \left(\cfrac{x-x_c}{f}\right) + x_c$$

$$ y' = \left(\cfrac{y-y_c}{cos \left(\cfrac{x-x_c}{f}\right)}\right) + y_c$$

These equations describe a cylindrical projection, where f represents the focal length of the camera in pixels (typical values range from `100` to `500`). The original image coordinates (x, y) are transformed into cylindrical coordinates (x', y'). The center of the image is denoted by (x_c, y_c).

---

let $ M = \begin{bmatrix} y_{1,1} & x_{1,1} & y_{1,2} & x_{1,2} \ y_{2,1} & x_{2,1} & y_{2,2} & x_{2,2} \ \vdots & \vdots & \vdots & \vdots \ y_{n,1} & x_{n,1} & y_{n,2} & x_{n,2} \end{bmatrix} M \in \mathbb{Z}^{n \times 4}$ be a set of matches

let $H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix}$ transform points from $M\left[y_{i,1}, x_{i,1}, 1\right) \rightarrow M\left[y_{i,2}, x_{i,2}, 1\right)$

Let $X = \begin{bmatrix} y_{1,1} & x_{1,1} & 1\ y_{2,1} & x_{2,1} & 1\ \vdots & \vdots & \vdots \ y_{n,1} & x_{n,1} & 1\ \end{bmatrix} $ and Let $Y = \begin{bmatrix} y_{1,2} & x_{1,2} & 1\ y_{2,2} & x_{2,2} & 1\ \vdots & \vdots & \vdots \ y_{n,2} & x_{n,2} & 1\ \end{bmatrix} X,Y \in \mathbb{Z}^{n \times 3} $

H is obtained by the closed form solution to linear regression $H = \left(X^T X\right)^{-1} X^T Y$

H transforms points via $=\begin{bmatrix} y_{i,2} \ x_{i,2} \ 1 \end{bmatrix} = H\begin{bmatrix} y_{i,1} \ x_{i,1} \ 1 \end{bmatrix} = \begin{bmatrix} h_{1,1}(y_{i,1}) + h_{1,2}(x_{i,1}) + h_{1,3}(1) \ h_{2,1}(y_{i,1}) + h_{2,2}(x_{i,1}) + h_{2,3}(1) \ h_{3,1}(y_{i,1}) + h_{3,2}(x_{i,1}) + h_{3,3}(1) \ \end{bmatrix} $

Let $ C_1 = \begin{bmatrix} y_1 & x_1 & 1 \ y_2 & x_2 & 1 \ \vdots & \vdots & \vdots \ y_n & x_n & 1\ \end{bmatrix} $ be a set of points to be mapped from img 1 to img 2

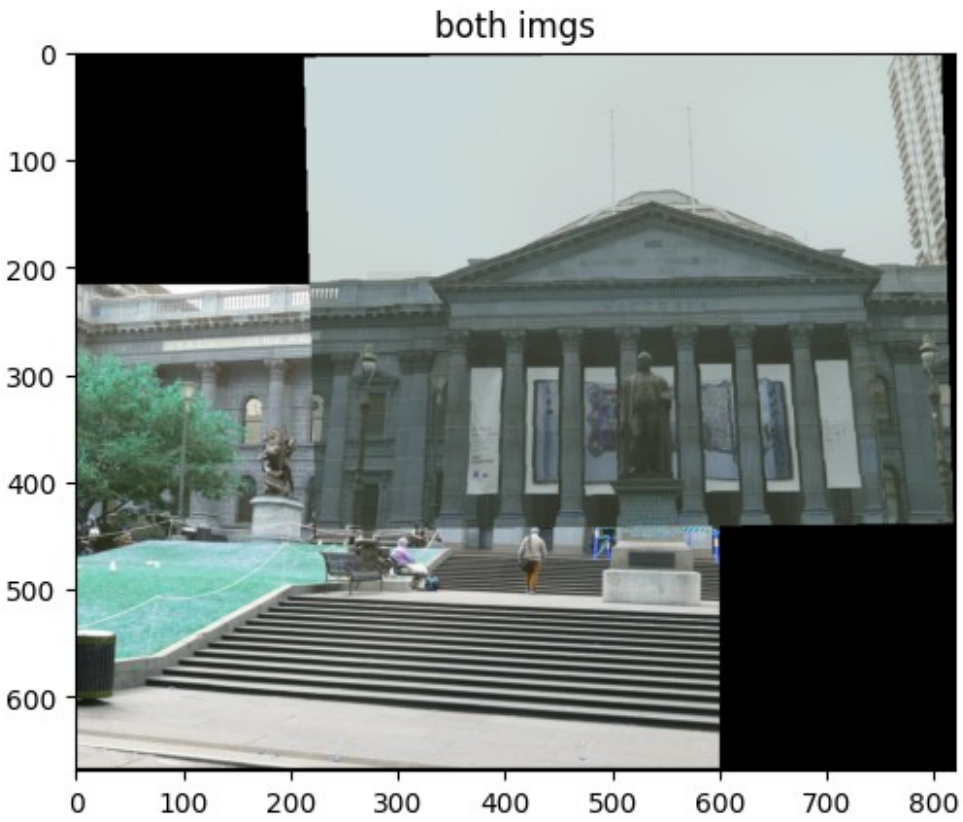
we can transform the set in one shot with CH $ \begin{bmatrix} h_{1,1}(y_1) + h_{1,2}(x_1) + h_{1,3}(1) & h_{2,1}(y_1) + h_{2,2}(x_1) + h_{2,3}(1) & h_{3,1}(y_1) + h_{3,2}(x_1) + h_{3,3}(1) \ h_{1,1}(y_2) + h_{1,2}(x_2) + h_{1,3}(1) & h_{2,1}(y_2) + h_{2,2}(x_2) + h_{2,3}(1) & h_{3,1}(y_2) + h_{3,2}(x_2) + h_{3,3}(1) \ \vdots & \vdots & \vdots \ h_{1,1}(y_n) + h_{1,2}(x_n) + h_{1,3}(1) & h_{2,1}(y_n) + h_{2,2}(x_n) + h_{2,3}(1) & h_{3,1}(y_n) + h_{3,2}(x_n) + h_{3,3}(1) \end{bmatrix} $

```
def warp_and_blend(img1, img2, H, display=False, blend = True):
    #show_img(img1, title=f"img1: {img1.shape}")
    #show_img(img2, title=f"img2: {img2.shape}")
```

```
    H_inv = np.linalg.inv(H)

    # Get the height and width of the two images
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # Corners of img2
    corners = np.array([[0, 0, 1], [h2, 0, 1], [0, w2, 1], [h2, w2,
1]])
    #print(f"corners: {corners.shape}")
    #print(corners)
    #print()

    transformed_corners = corners @ H_inv
    #print(f"transformed_corners.shape: {transformed_corners.shape}")
    #print(transformed_corners)
    #print()

    #find if we need to translate img 1
    #if none of the corners in img2 have negative indicies then we
dont need to translate
    transformed_corners_min_y = np.min(transformed_corners[:, 0])
    transformed_corners_min_x = np.min(transformed_corners[:, 1])

    transformed_corners_max_y = np.max(transformed_corners[:, 0])
    transformed_corners_max_x = np.max(transformed_corners[:, 1])

    min_y = np.min([transformed_corners_min_y, 0])
    #print(f"min_y: {min_y}")
    min_x = np.min([transformed_corners_min_x, 0])
    #print(f"min_x: {min_x}")


    #transformation from img1 to new_img is (y,x) -> (y+|min(y)|,x+|
min(y)|)
    y_offset = np.abs(min_y)
    x_offset = np.abs(min_x)
    #print(f"y_offset: {y_offset}")
    #print(f"x_offset: {x_offset}")


    #what is the furthest pixel from the origin on either axis
    max_y = np.max([transformed_corners_max_y, h1])
    #print(f"max_y: {max_y}")
    max_x = np.max([transformed_corners_max_x, w1])
    #print(f"max_x: {max_x}")


    padding = 1
    new_img_height = max_y - min_y + padding
```

```
    new_img_height = np.ceil(new_img_height).astype(int)


    new_img_width = max_x - min_x + padding
    new_img_width = np.ceil(new_img_width).astype(int)


    #print(f"{max_y}-{min_y} new_img_height: {new_img_height}")
    #print(f"{max_x}-{min_x} new_img_width: {new_img_width}")
    new_img = np.zeros((new_img_height, new_img_width, 3),
dtype=np.uint8)
    #print(f"new_img: {new_img.shape}")



    img1_new_origin_y = 0 + y_offset
    img1_new_origin_y = np.floor(img1_new_origin_y).astype(int)
    img1_new_origin_x = 0 + x_offset
    img1_new_origin_x = np.floor(img1_new_origin_x).astype(int)

    #print(f"img1_new_origin_y:{img1_new_origin_y}")
    #print(f"img1_new_origin_x:{img1_new_origin_x}")


    new_img[img1_new_origin_y:img1_new_origin_y+h1,
img1_new_origin_x:img1_new_origin_x+w1] = img1
    #show_img(new_img, title = "just img 1")

    # Generate the coordinate grids
    y_coords, x_coords = np.meshgrid(np.arange(h2), np.arange(w2),
indexing='ij')

    # Create a homogeneous coordinate grid (all ones)
    ones = np.ones_like(y_coords)

    # Stack to create a n x 3 array
    points = np.stack((y_coords, x_coords, ones), axis=-1)

    # Reshape the points into an n x 3 array (flattening the grid)
    points = points.reshape(-1, 3)  # Reshape to (h2 * w2, 3)
    #print(f"points: {points.shape}")
    #print(points)
    #print()

    transformed_points = points @ H_inv
    transformed_points[:,0] += y_offset
    transformed_points[:,1] += x_offset
    #print(f"transformed_points: {transformed_points.shape}")
    #print(transformed_points)
    #print()
    transformed_points = np.round(transformed_points).astype(int)
```

```
    #print(f"transformed_points: {transformed_points.shape}")
    #print(transformed_points)
    #print()

    if blend:
      mean1, std1 = cv2.meanStdDev(img1)
      mean2, std2 = cv2.meanStdDev(img2)

      # Reshape mean and std to be broadcastable to the image shape
      mean1 = mean1.reshape(1, 1, 3)  # Reshape from (3, 1) to (1, 1,
3)
      std1 = std1.reshape(1, 1, 3)
      mean2 = mean2.reshape(1, 1, 3)
      std2 = std2.reshape(1, 1, 3)

      img2 = ((img2 - mean2) * (std1 / std2)) + mean1

      img2 = np.clip(img2, 0, 255).astype(np.uint8)

    new_img[transformed_points[:,0], transformed_points[:,1], :] =
img2[points[:,0], points[:,1]]

    """
    #show_img(new_img, title="just image 1")
    for row in range(img2.shape[0]):
      for col in range(img2.shape[1]):
        new_point = np.array([row, col, 1]) @ H_inv
        new_point = np.round(new_point).astype(int)
        #print(f"({row},{col})->{new_point}")
        new_img[new_point[0]+y_offset, new_point[1]+x_offset] =
img2[row, col]
    """
    if display:
      show_img(new_img, title = "both imgs")

    return new_img
if test_each_cell:
  warp_and_blend(img1 = img_array[1], img2=img_array[2], H=homo,
display=True)
```



Note: The pipeline talks about how to stitch a pair of images, you need to extend this to work for multiple images. You can re-run your images pairwise or do something smarter.

Your end goal is to be able to stitch any number of given images - maybe 2 or 3 or 4 or 100, your algorithm should work. If a random image with no matches are given, your algorithm needs to report an error.

Note: When blending these images, there are inconsistency between pixels from different input images due to different exposure/white balance settings or photometric distortions or vignetting. This can be resolved by algorithms such as Poisson blending. You can use third party code for the seamless panorama stitching.

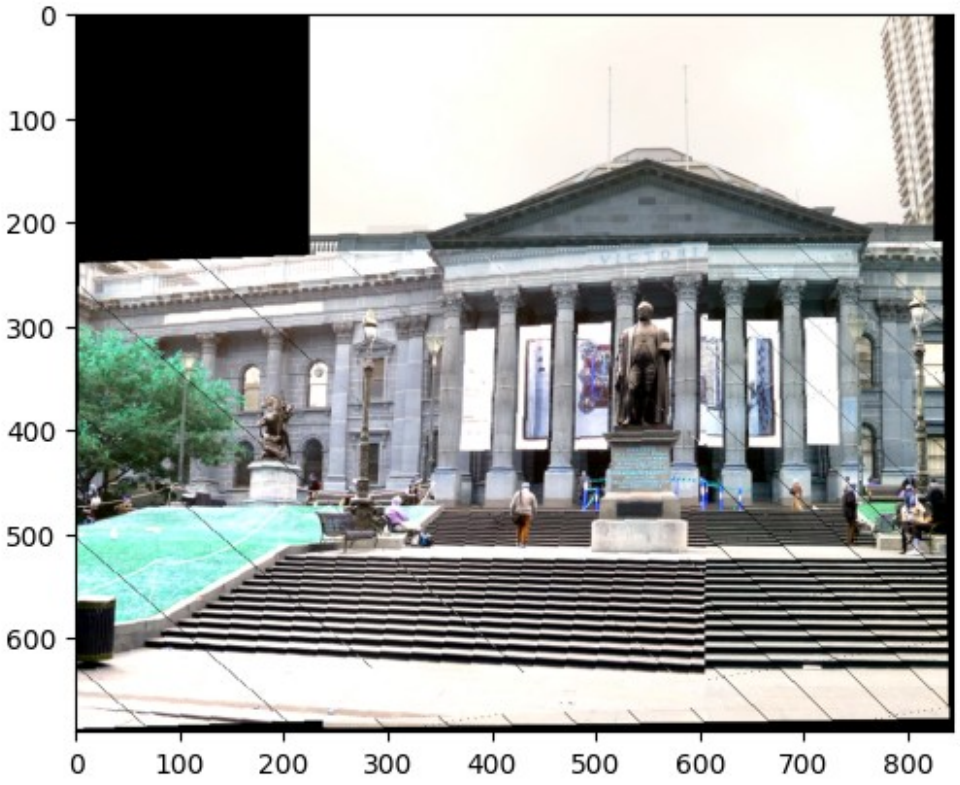
Final Stage: Putting Everything Together! [5 pts in total]

```
# Take in a list of images and stitch them together!
def mypano(img_list, num_strong_corners=2000, num_best_corners = 1000,
           feature_patch_side_length = 40, feature_Blur_Kernel =
(3,3), feature_subsample_side_length = 16,
           match_ratio_threshold = 0.75, ransac_iter = 10000,
ransac_sample_size = 4,
           display = False, feature_descriptor_method="mine"):
```

```
  # Use this function as a wrapper and call all the functions that you
wrote here!
  # Feel free to use for loops to call functions (like corner
detection) multiple times
  # 1. Load Images
  img_array = []
  for f in img_list:
    img = cv2.imread(f)
    img_array.append(img)

  working_img = img_array[0]
  if feature_descriptor_method == "sift":
    #https://docs.opencv.org/4.x/d7/d60/classcv_1_1SIFT.html
    sift = cv2.SIFT_create(nfeatures = num_best_corners)

  for i in range(1, len(img_array)):
    show_img(working_img, title=f"working img {i}") if display else
None
    img_i = img_array[i]

    grey_img_i = cv2.cvtColor(img_i, cv2.COLOR_BGR2GRAY)
    working_grey_img = cv2.cvtColor(working_img, cv2.COLOR_BGR2GRAY)


    if feature_descriptor_method == "mine":
      # 2. Detect Corners
      strong_corners_i, cmap_i = detect_corner(img_i,
n=num_strong_corners, display_corners = display, display_title =
f"img{i} {num_strong_corners} strong corners")
      working_strong_corners, working_cmap =
detect_corner(working_img, n=num_strong_corners, display_corners =
display, display_title = f"working{i} {num_strong_corners} strong
corners")


      # 3. ANMS
      best_corners_i = anms(cmap_i, strong_corners_i,
num_best_corners, display_corners = display, display_title = f"img{i}
{num_best_corners} best corners", img=img_i)
      working_best_corners = anms(working_cmap,
working_strong_corners, num_best_corners, display_corners = display,
display_title = f"working{i} {num_best_corners} best corners",
img=working_img)
    elif feature_descriptor_method == "sift":
      # Detect keypoints
      best_corners_i = sift.detect(grey_img_i, None)
      working_best_corners = sift.detect(working_grey_img, None)

      if display:
        image_i_with_keypoints = cv2.drawKeypoints(grey_img_i,
```

```
best_corners_i, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        working_image_with_keypoints =
cv2.drawKeypoints(working_grey_img, working_best_corners, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        show_img(image_i_with_keypoints, title=f"img{i} sift
keypoints")
        show_img(working_image_with_keypoints, title=f"working{i} sift
keypoints")

      #print(f"best_corners_i: {best_corners_i}")
      #print(f"best_corners_i: {best_corners_i.shape}")
      #print(f"working_best_corners: {working_best_corners}")
      #print(f"working_best_corners: {working_best_corners.shape}")
    else:
      raise ValueError(f"Invalid feature descriptor method:
{feature_descriptor_method}")

    # 4. Feature Descriptors
    if feature_descriptor_method == "mine":
      desc_i, des_coords_i = feature_desc(grey_img_i, best_corners_i,
display_descriptors=display, display_title = f"feature descriptors:
{i}", patch_side_length = feature_patch_side_length, Blur_Kernel =
feature_Blur_Kernel, subsample_side_length =
feature_subsample_side_length)

      working_desc, working_des_coord = feature_desc(working_grey_img,
working_best_corners,  display_descriptors=display, display_title =
f"feature descriptors: working", patch_side_length =
feature_patch_side_length, Blur_Kernel = feature_Blur_Kernel,
subsample_side_length = feature_subsample_side_length)
    elif feature_descriptor_method == "sift":
      #print(f"dir(sift): {dir(sift)}")
      #sift.compute(img, keypoints)
      des_coords_i, desc_i = sift.compute(grey_img_i, best_corners_i)

      working_des_coord, working_desc = sift.compute(working_grey_img,
working_best_corners)

      des_coords_i = np.array([[corner.pt[1], corner.pt[0]] for corner
in des_coords_i]).astype(int)
      working_des_coord = np.array([[corner.pt[1], corner.pt[0]] for
corner in working_des_coord]).astype(int)


      #print(f"des_coords_i: {des_coords_i}")
      #print(f"desc_i: {desc_i}")

      #print(f"working_des_coord: {working_des_coord}")
      #print(f"working_desc: {working_desc}")
```

```
      #print(f"des_coords_i: {des_coords_i.shape}")
      #print(f"desc_i: {desc_i.shape}")

      #print(f"working_des_coord: {working_des_coord.shape}")
      #print(f"working_desc: {working_desc.shape}")


    # 5. Feature Matching
    matches = feature_matching(working_desc, desc_i,
working_des_coord, des_coords_i, img1 = working_img, img2 = img_i,
display = display, ratio_threshold = match_ratio_threshold)
    #print(matches.shape)
    # 6. RANSAC/Refine Matches
    homo, inliers = RANSAC(matches, iter = ransac_iter, N = 0.9 *
matches.shape[0], thr = 2, sample_size = ransac_sample_size, img1 =
working_img, img2=img_i, display = display)

    # 7. Warp and Blend
    working_img = warp_and_blend(img1 = working_img, img2=img_i,
H=homo, display=display)

  # Return Stitched Image
  return working_img

if test_each_cell:
  img_paths =
['/content/VictoriaLibrary/1.jpg','/content/VictoriaLibrary/2.jpg','/
content/VictoriaLibrary/3.jpg']
  img = mypano(img_paths, num_strong_corners=2000, num_best_corners =
1000, display = False)
  show_img(img)
```

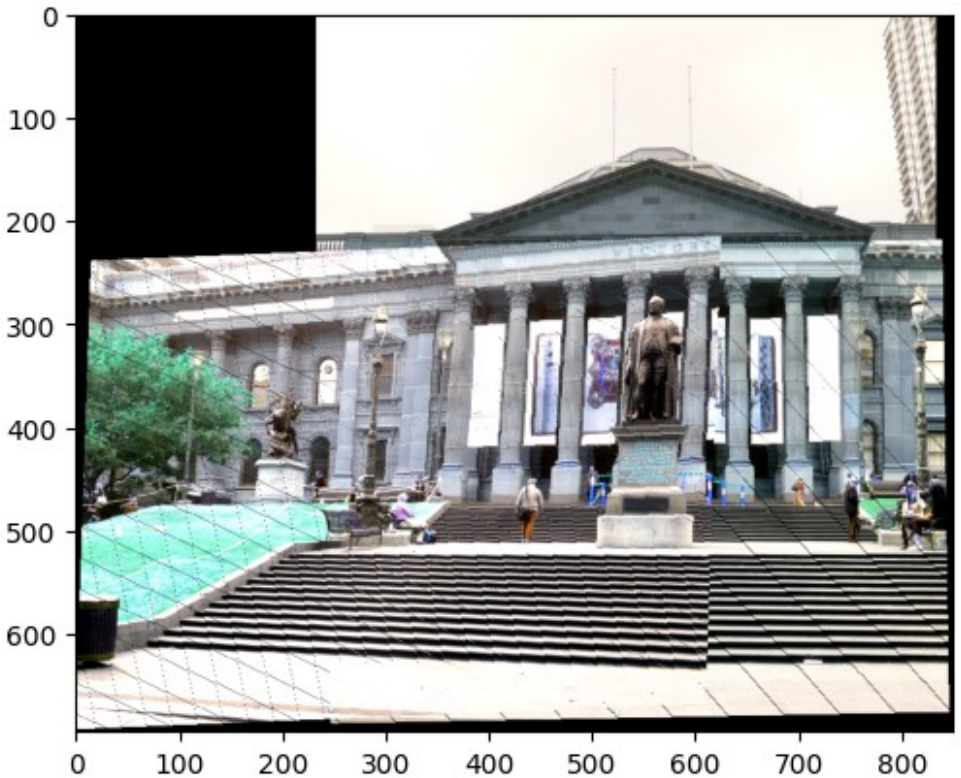

0
100
200
300
400
500
600
0
100
200
300
400
500
600
700
800

Now perform the same task for other datasets. [20 pts in total]

Set 1 [5pts]: VictoriaLibrary

```
#import sys
#np.set_printoptions(threshold=sys.maxsize)
# VictoriaLibrary
files = gdown.download_folder(id="14lVfW-ss6qo24PvcVdlBNW0o40u6QlOw",
quiet=True, use_cookies=False)
files = ['/content/VictoriaLibrary/1.jpg',
'/content/VictoriaLibrary/2.jpg', '/content/VictoriaLibrary/3.jpg']

print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = False)
show_img(img)
```

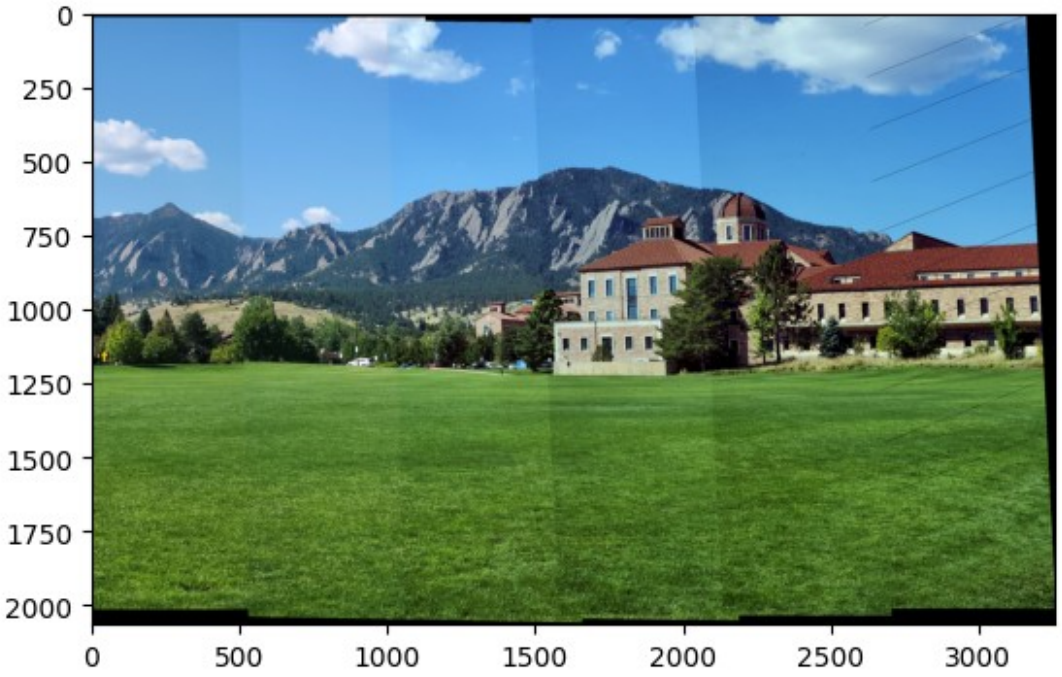
```
files: ['/content/VictoriaLibrary/1.jpg',
'/content/VictoriaLibrary/2.jpg', '/content/VictoriaLibrary/3.jpg']
```



Set 2 [5pts] (Flatirons)

```
# Flatirons
files = gdown.download_folder(id="1cwLOG_4keUaQHmp5RVwe_ZiAHeULpjOd",
quiet=True, use_cookies=False)[:-3]
files = ['/content/Flatirons/1.jpg', '/content/Flatirons/2.jpg',
'/content/Flatirons/3.jpg', '/content/Flatirons/4.jpg',
'/content/Flatirons/5.jpg']
print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = False)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
show_img(img)
```

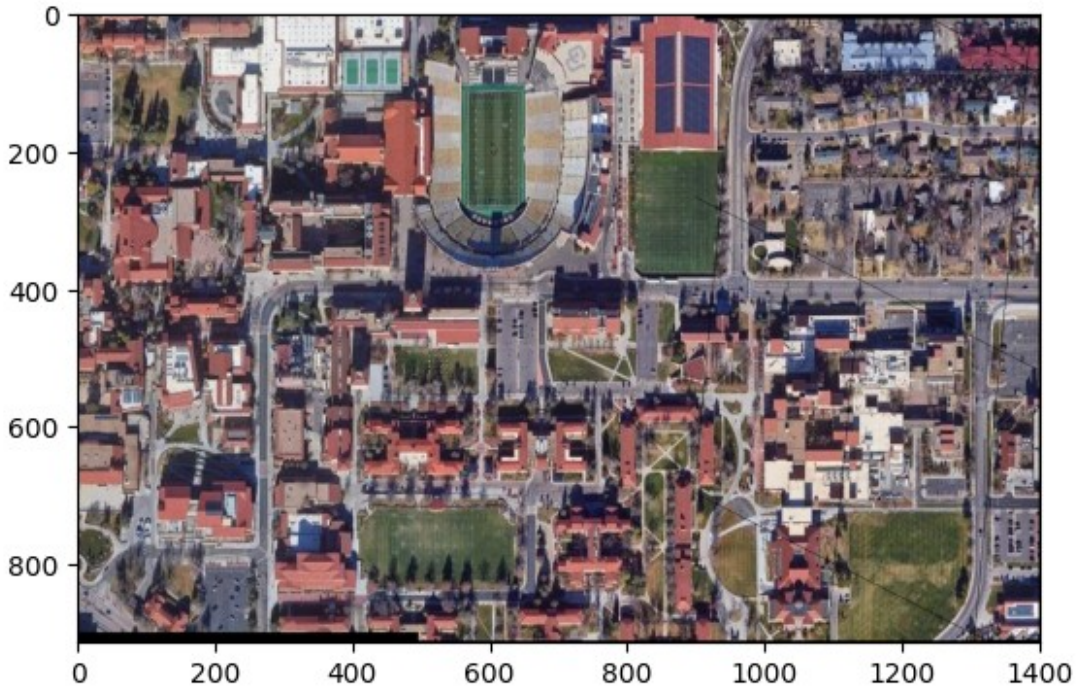
```
files: ['/content/Flatirons/1.jpg', '/content/Flatirons/2.jpg',
'/content/Flatirons/3.jpg', '/content/Flatirons/4.jpg',
'/content/Flatirons/5.jpg']
```



Set 3 [5pts] (CUBoulderSatView)

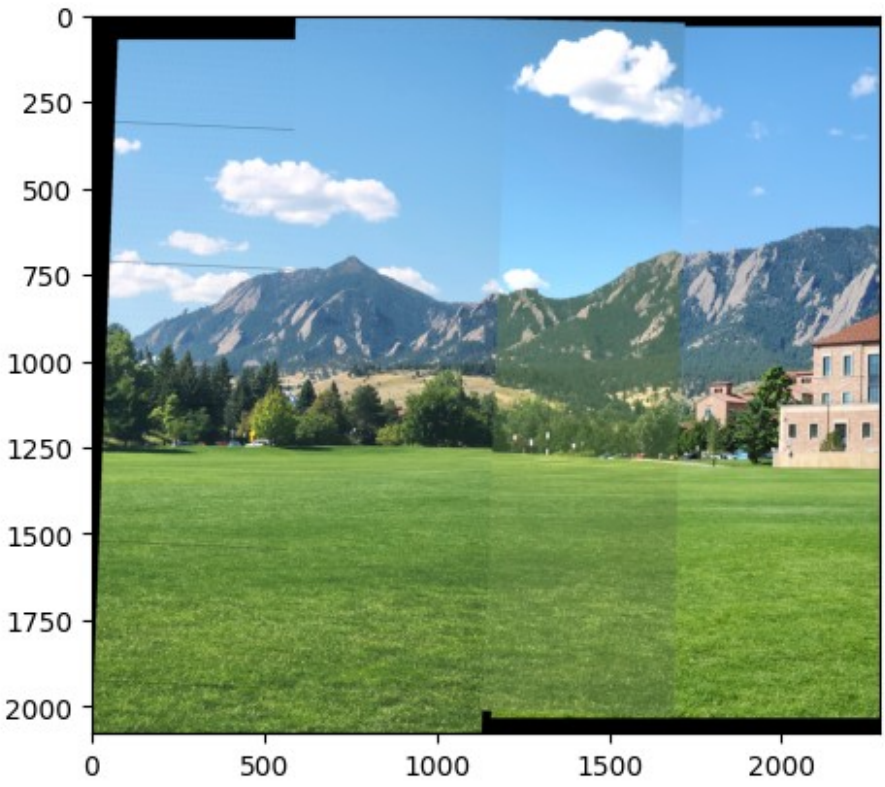
```
# CUBoulderSatView
files = gdown.download_folder(id="10WcuZFuCxbe1FQSwpdy_9piyMM45eOI_",
quiet=True, use_cookies=False)
files = ['/content/CUBoulderSatView/1.jpg',
'/content/CUBoulderSatView/2.jpg']
print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = False)
```

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
show_img(img)

files: ['/content/CUBoulderSatView/1.jpg',
'/content/CUBoulderSatView/2.jpg']
```



Set 4 [5pts] (FlatironsChallenge)

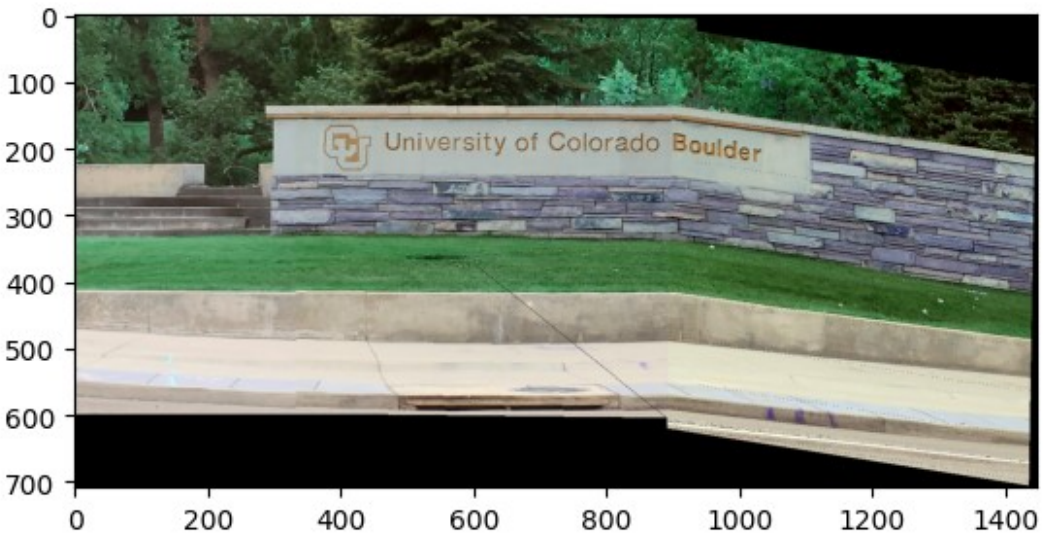
```
# FlatironsChallenge
files = gdown.download_folder(id="16_AV0IAfdr594jE_rK7serWNsVeN7wCS",
quiet=True, use_cookies=False)
files = ['/content/FlatironsChallenge/1.jpg',
'/content/FlatironsChallenge/2.jpg',
'/content/FlatironsChallenge/3.jpg']

print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = False)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
show_img(img)

files: ['/content/FlatironsChallenge/1.jpg',
'/content/FlatironsChallenge/2.jpg',
'/content/FlatironsChallenge/3.jpg']
```



Set 4 [5pts] (CULogo)

```
# CULogo
files = gdown.download_folder(id="1Pv7eI-LOOUZxHffz1D5pmMKOk-Y7C-kx",
quiet=True, use_cookies=False)
files = ['/content/CULogo/1.jpg', '/content/CULogo/2.jpg',
'/content/CULogo/3.jpg']
print(f"files: {files}")
#img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = False, feature_descriptor_method="mine", ransac_sample_size
= 4)
img = mypano(files, num_best_corners = 5000, display = False,
feature_descriptor_method="sift", ransac_sample_size = 4,
match_ratio_threshold=0.5)

show_img(img)
```

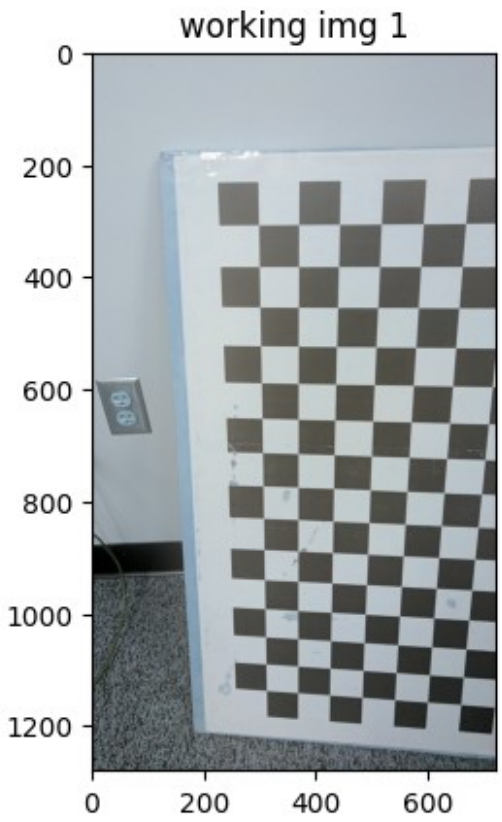
```
files: ['/content/CULogo/1.jpg', '/content/CULogo/2.jpg',
'/content/CULogo/3.jpg']
```

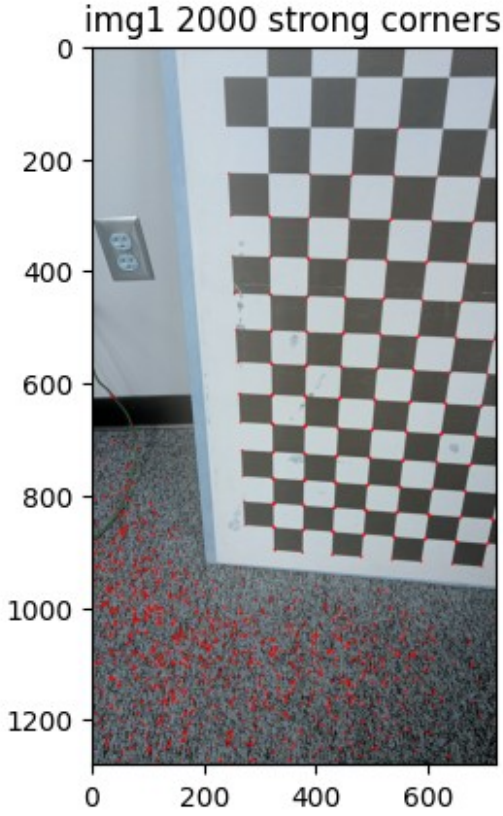


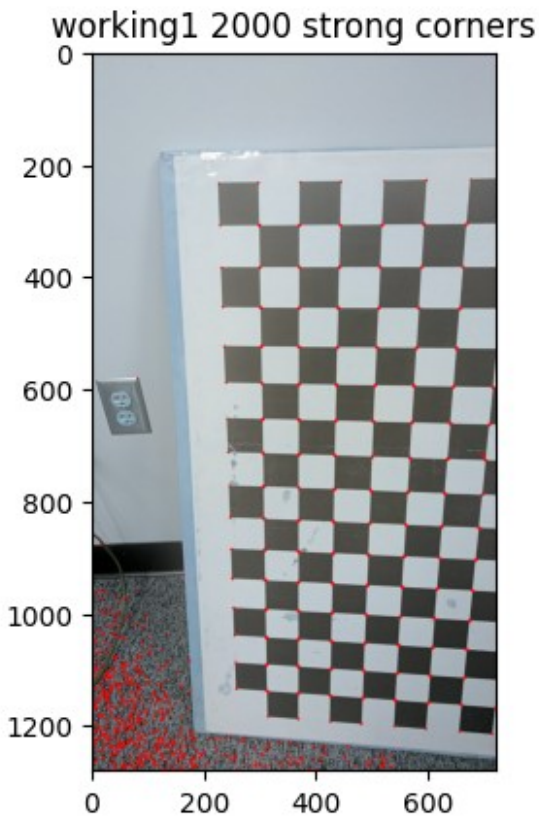
Set 5 [5pts] (Checkerboard)

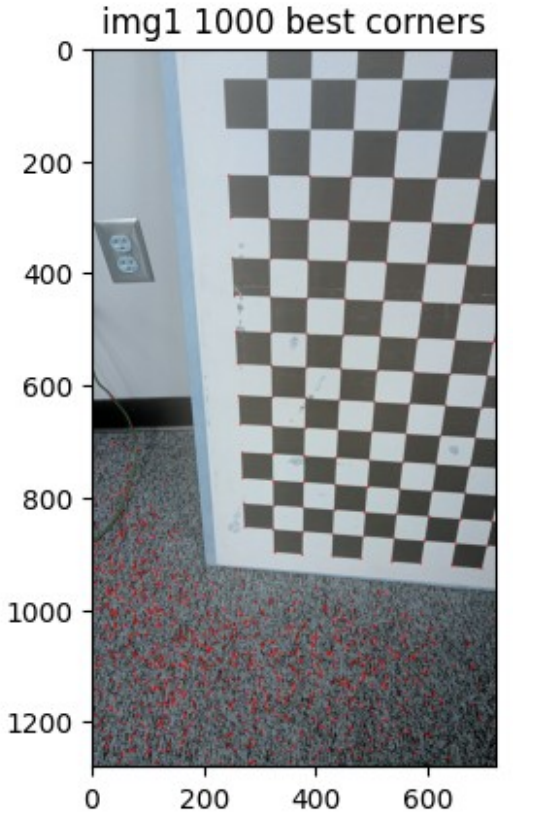
```
# Checkerboard0
files = gdown.download_folder(id="1xPzK5se5AICqffjfuedqgiovQse4ILkS",
quiet=True, use_cookies=False)
files = ['/content/Checkerboard/1.jpg', '/content/Checkerboard/2.jpg',
'/content/Checkerboard/3.jpg', '/content/Checkerboard/4.jpg']
print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = True,  feature_descriptor_method="mine")
#img = mypano(files, num_best_corners = 5000, display = True,
feature_descriptor_method="sift", ransac_sample_size = 4,
match_ratio_threshold=0.9, feature_patch_side_length=100,
feature_Blur_Kernel = (3,3), feature_subsample_side_length = 16)
show_img(img)
```

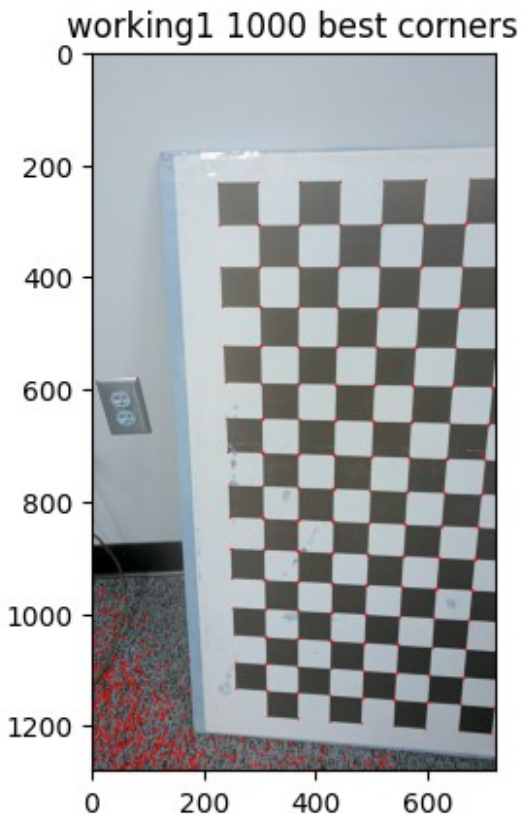
```
files: ['/content/Checkerboard/1.jpg', '/content/Checkerboard/2.jpg',
'/content/Checkerboard/3.jpg', '/content/Checkerboard/4.jpg']
```

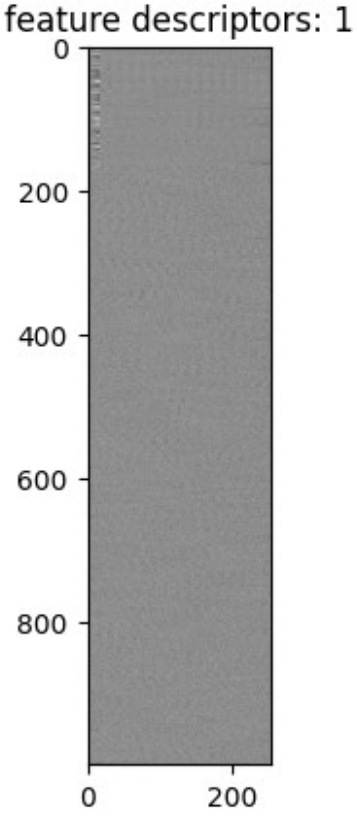

working img 1
0
200
400
600
800
1000
1200
0
200
400
600

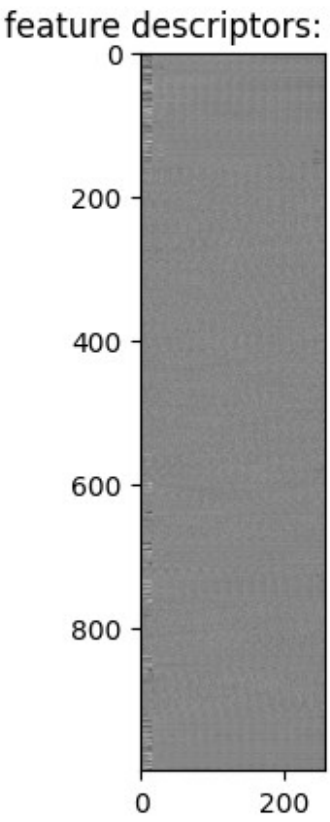

img1 2000 strong corners
0
200
400
600
800
1000
1200
0
200
400
600

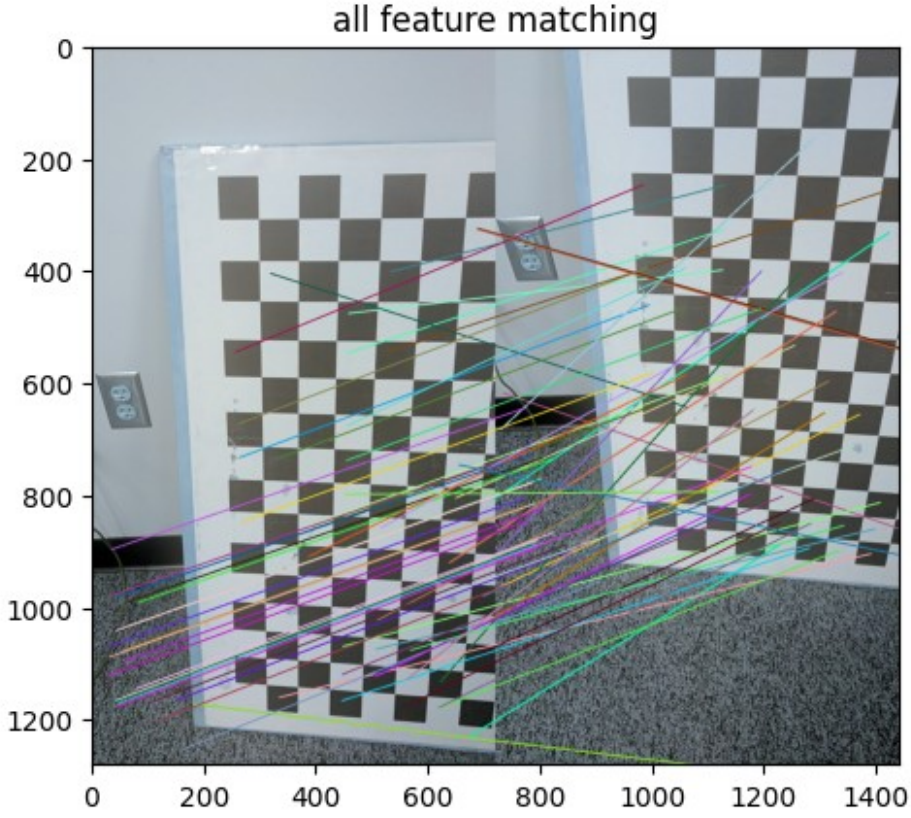

working1 2000 strong corners
0
200
400
600
800
1000
1200
0
200
400
600


img1 1000 best corners
0
200
400
600
800
1000
1200
0
200
400
600

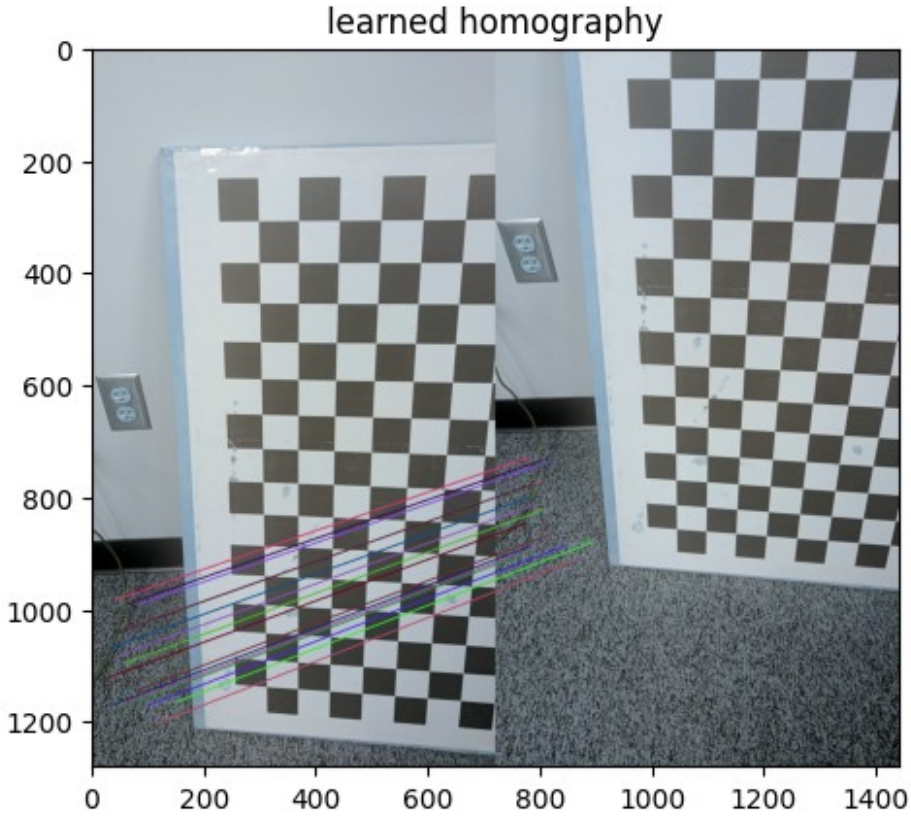

working1 1000 best corners
0
200
400
600
800
1000
1200
0
200
400
600

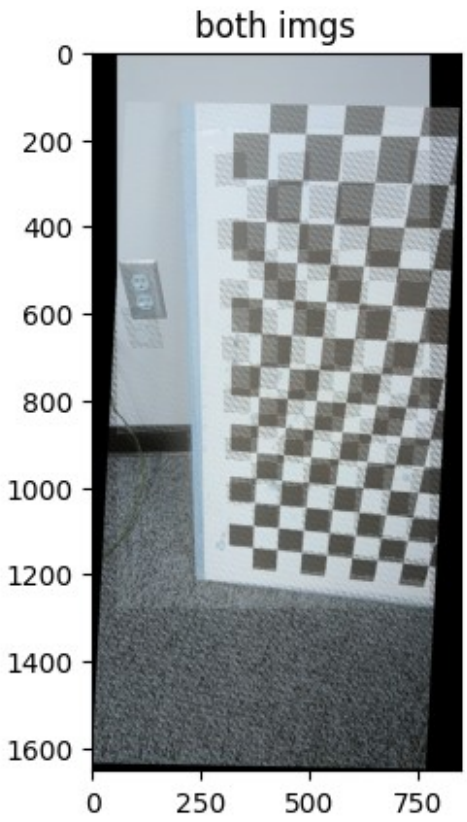

feature descriptors: 1
0
200
400
600
800
0
200

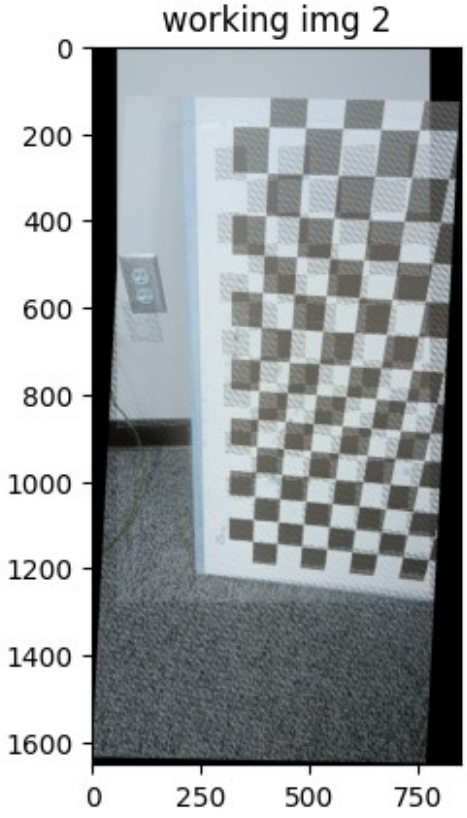

feature descriptors: working
0
200
400
600
800
0
200

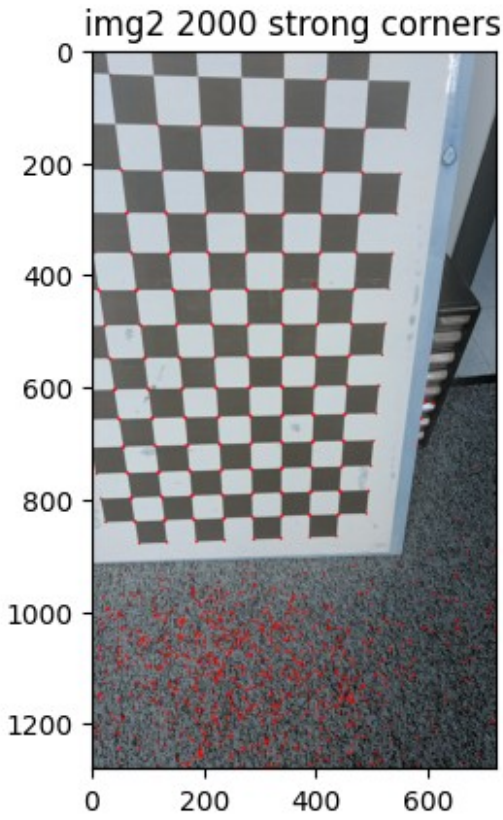

all feature matching
0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200
1400

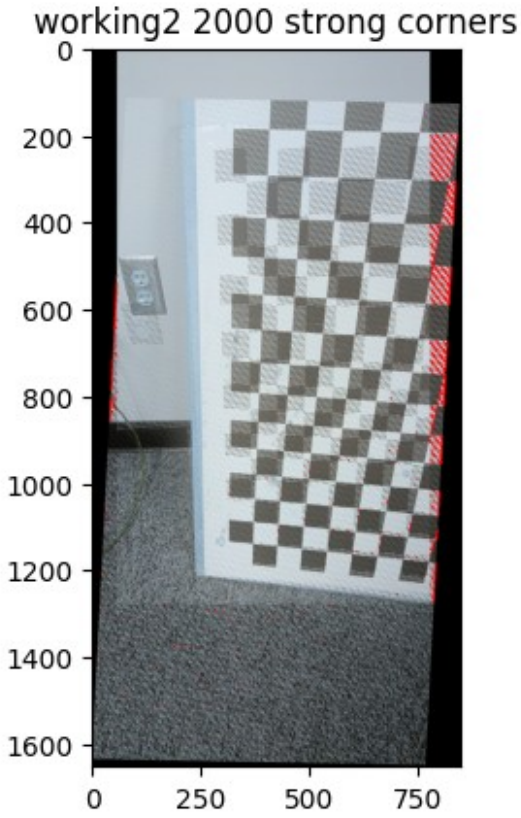
```
returning matrix with 16 inliers and mean error 117.17792102686326,
ranout of iterations
(16, 4)
```

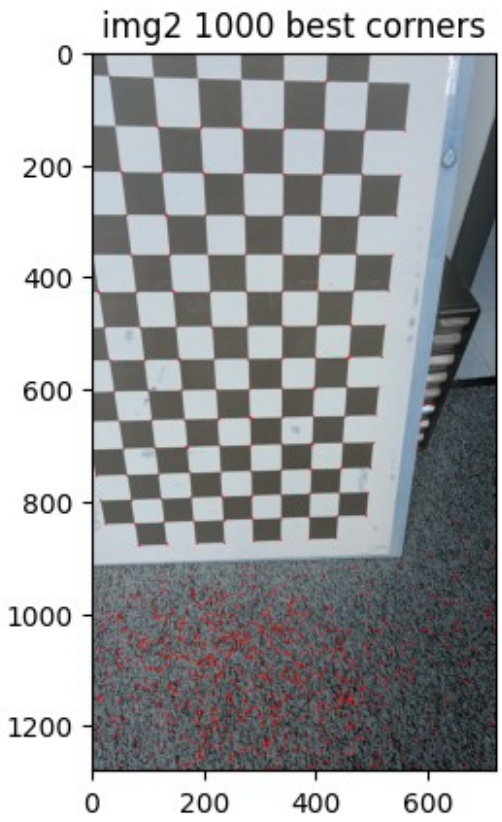


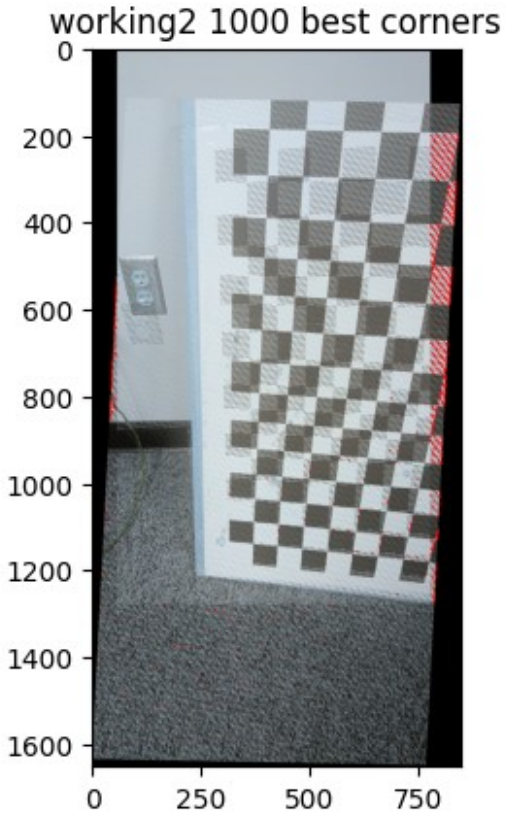

both imgs
0
200
400
600
800
1000
1200
1400
1600
0
250
500
750

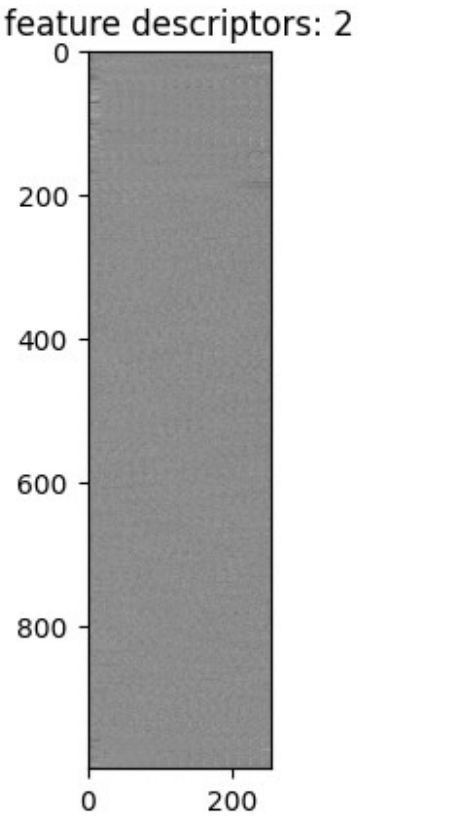

working img 2
0
200
400
600
800
1000
1200
1400
1600
0
250
500
750

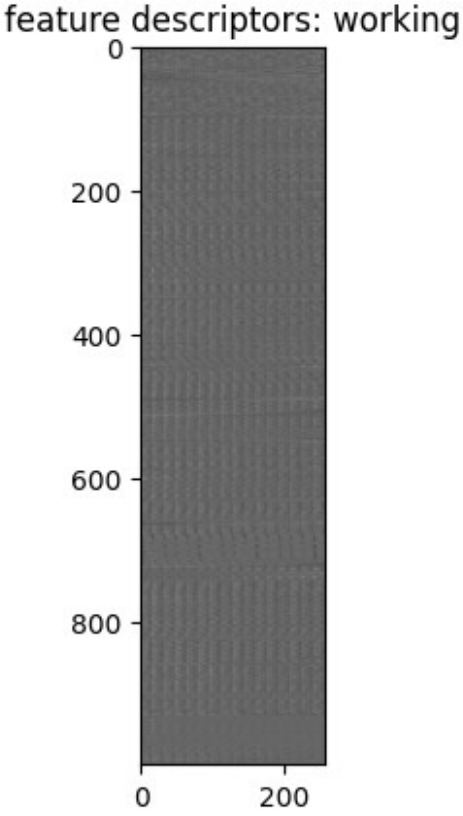

img2 2000 strong corners
0
200
400
600
800
1000
1200
0
200
400
600

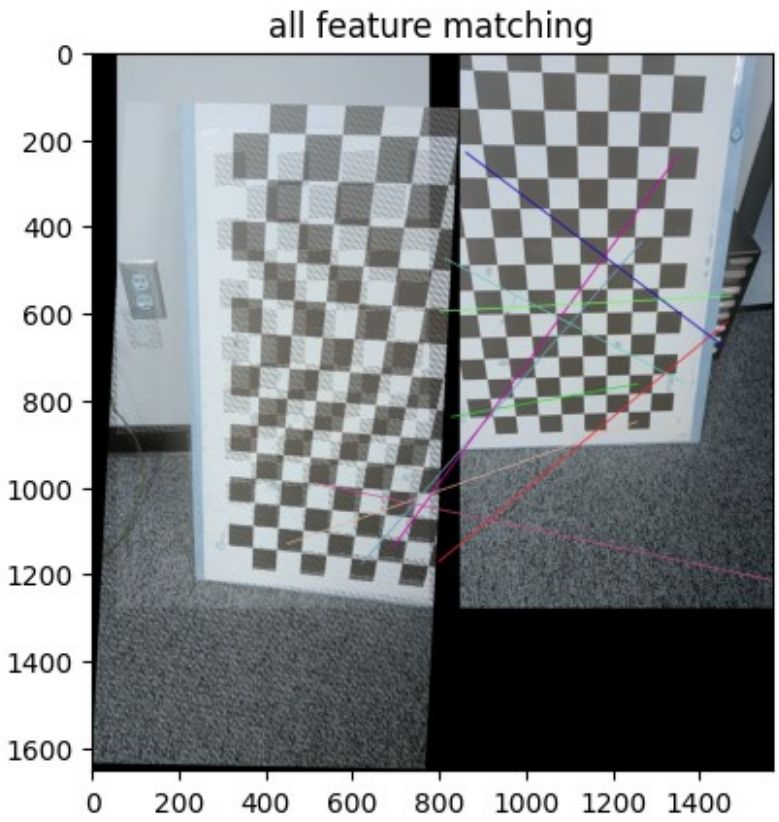

working2 2000 strong corners
0
200
400
600
800
1000
1200
1400
1600
0
250
500
750


img2 1000 best corners
0
200
400
600
800
1000
1200
0
200
400
600

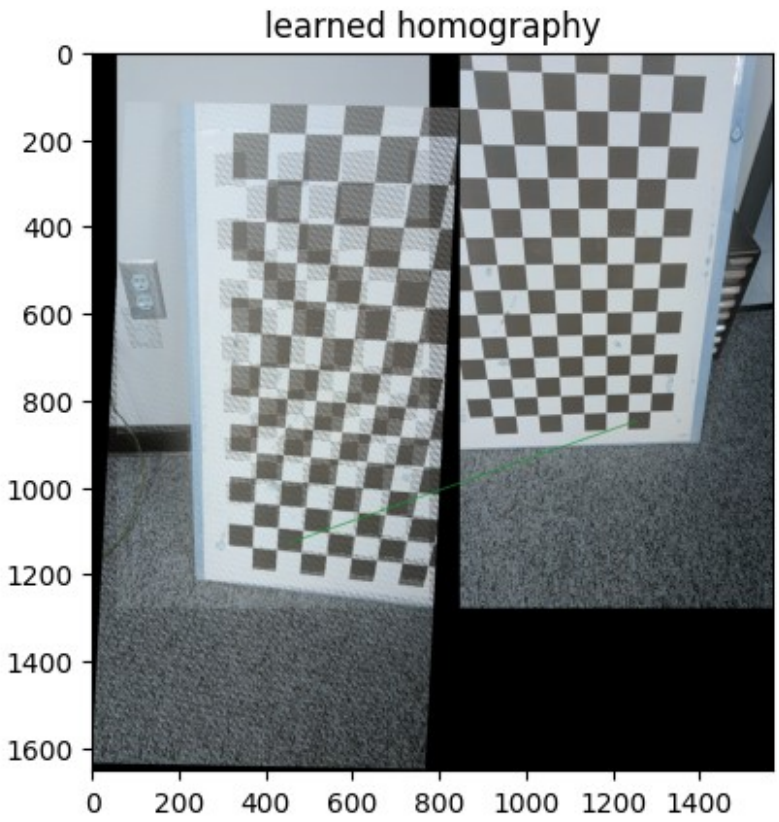

working2 1000 best corners
0
200
400
600
800
1000
1200
1400
1600
0
250
500
750

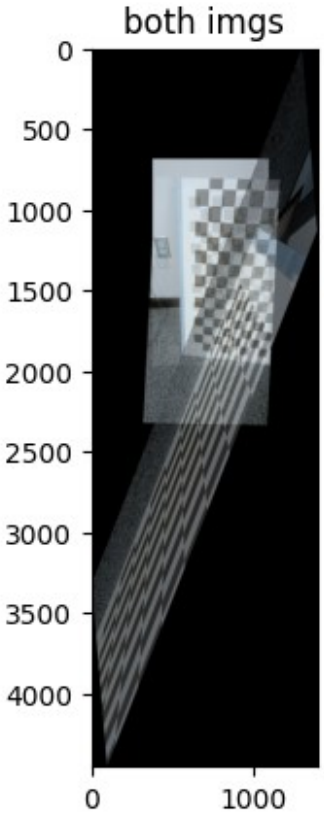

feature descriptors: 2
0
200
400
600
800
0
200

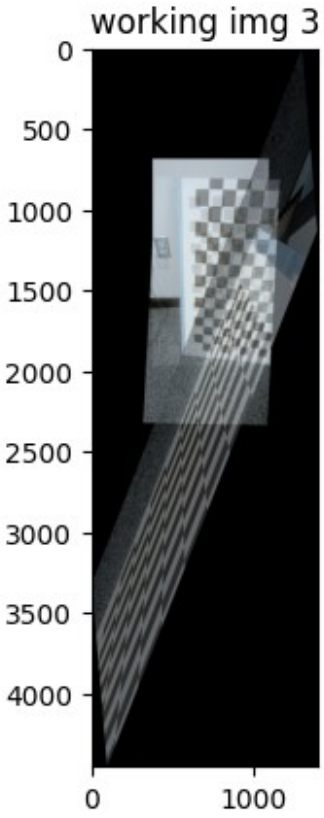

feature descriptors: working
0
200
400
600
800
0
200

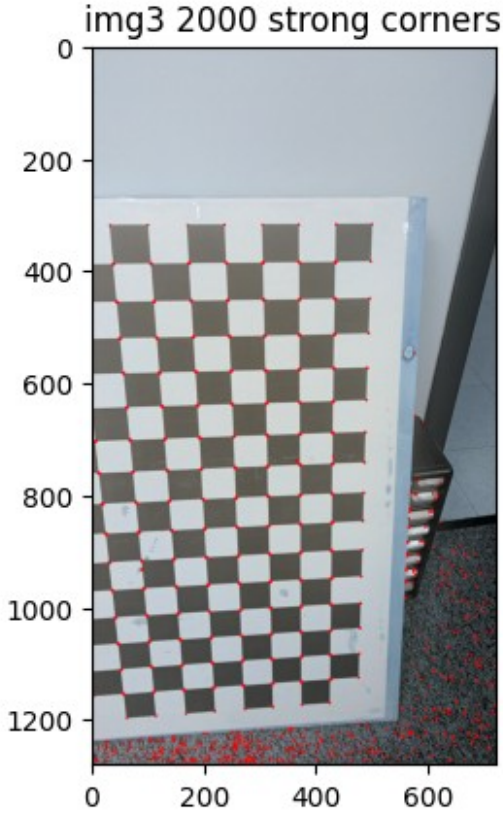


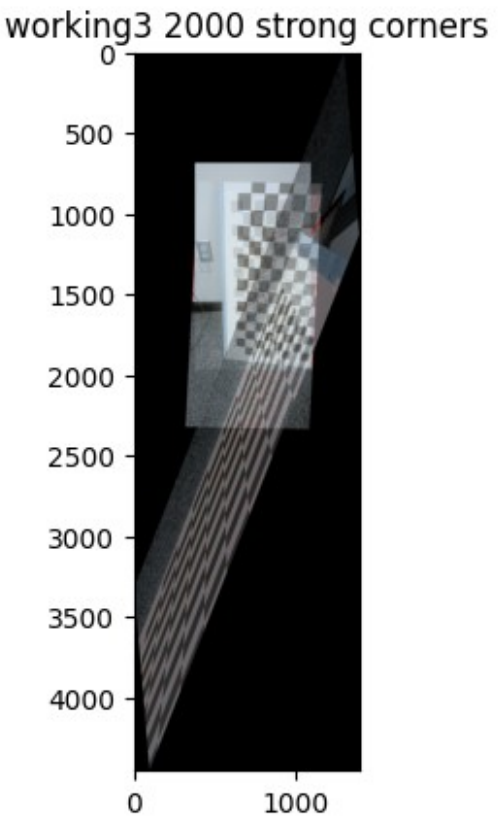

```
returning matrix with 1 inliers and mean error 234.6423119642376,
ranout of iterations
(1, 4)
```

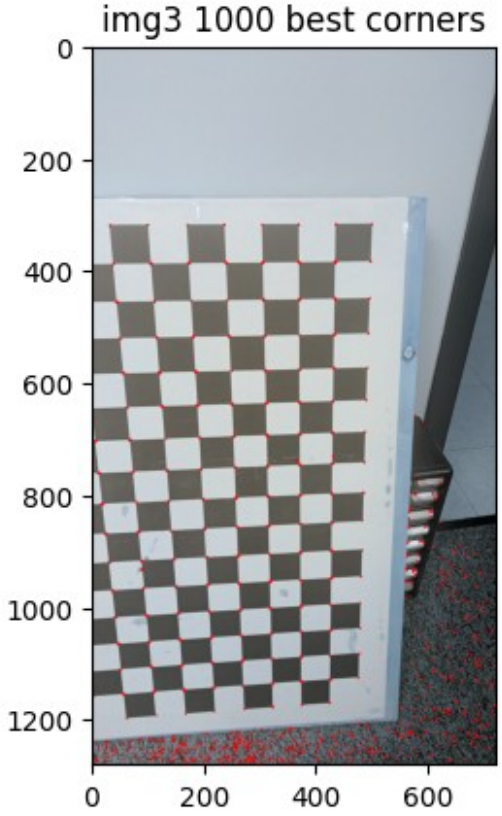

learned homography
0
200
400
600
800
1000
1200
1400
1600
0
200
400
600
800
1000
1200
1400


both imgs
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000

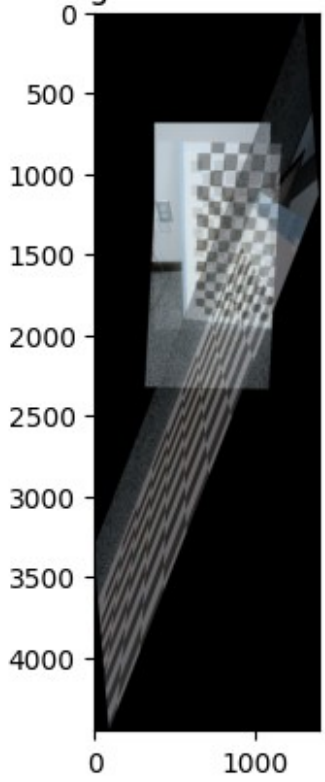

working img 3
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000


img3 2000 strong corners
0
200
400
600
800
1000
1200
0
200
400
600

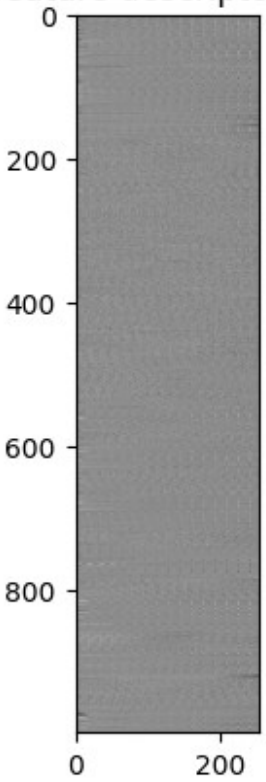

working3 2000 strong corners
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000

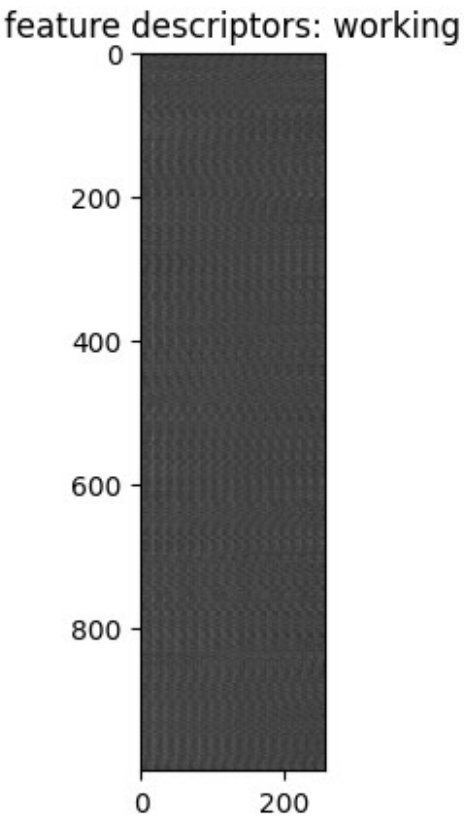

img3 1000 best corners
0
200
400
600
800
1000
1200
0
200
400
600

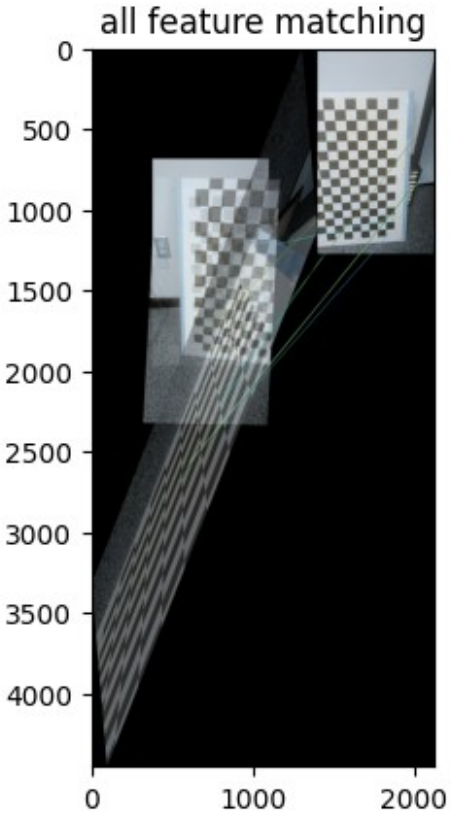
working3 1000 best corners



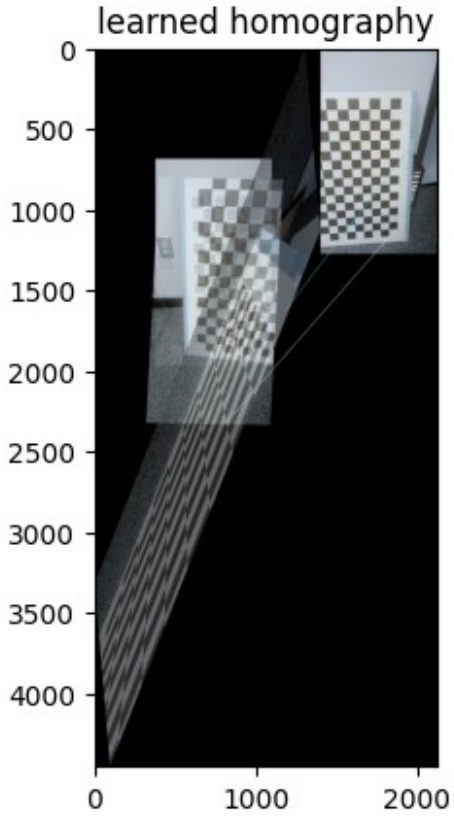
feature descriptors: 3

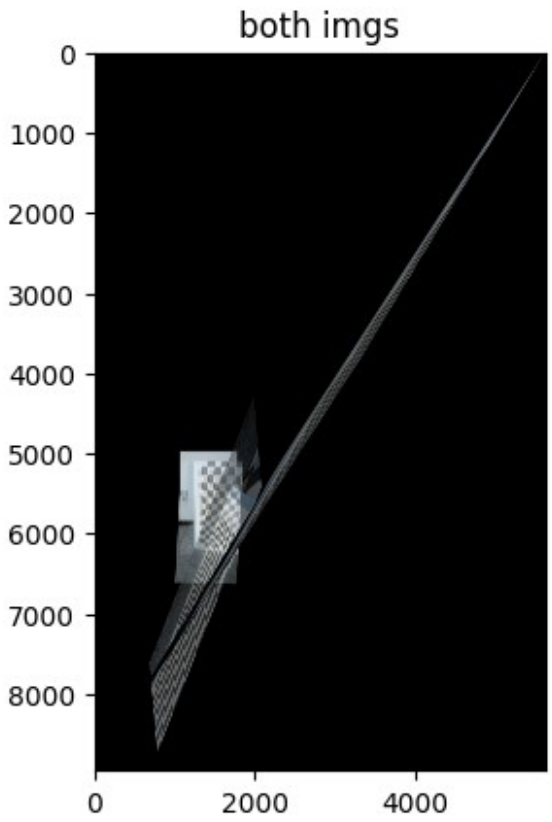


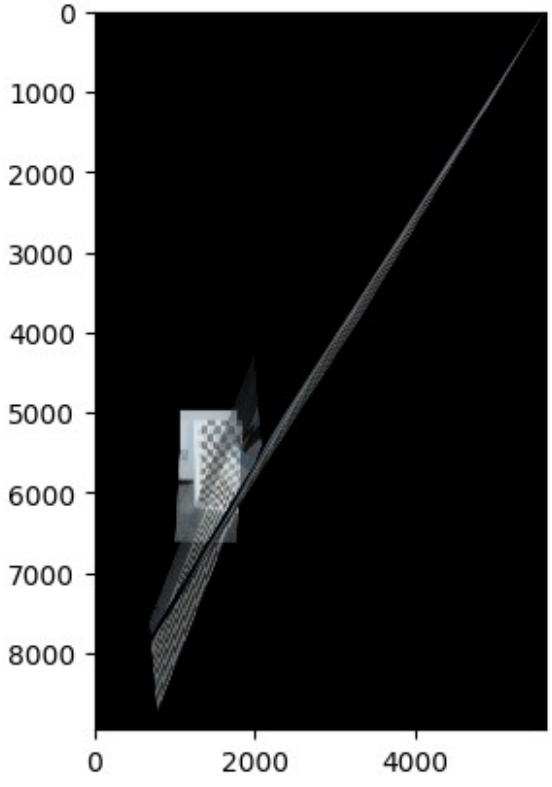

feature descriptors: working
0
200
400
600
800
0
200


all feature matching
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000

```
returning matrix with 2 inliers and mean error 1097.1334414568778,
ranout of iterations
(2, 4)
```



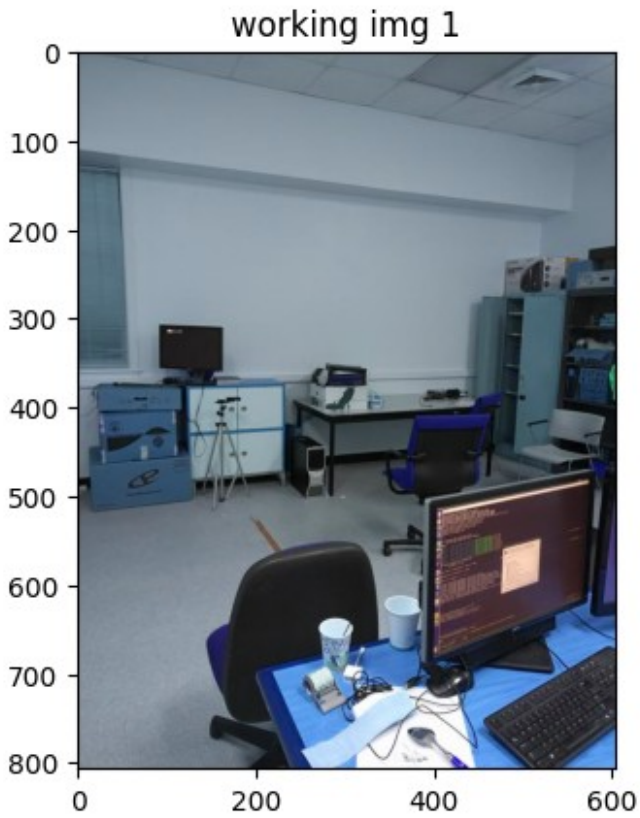

both imgs
0
1000
2000
3000
4000
5000
6000
7000
8000
0
2000
4000

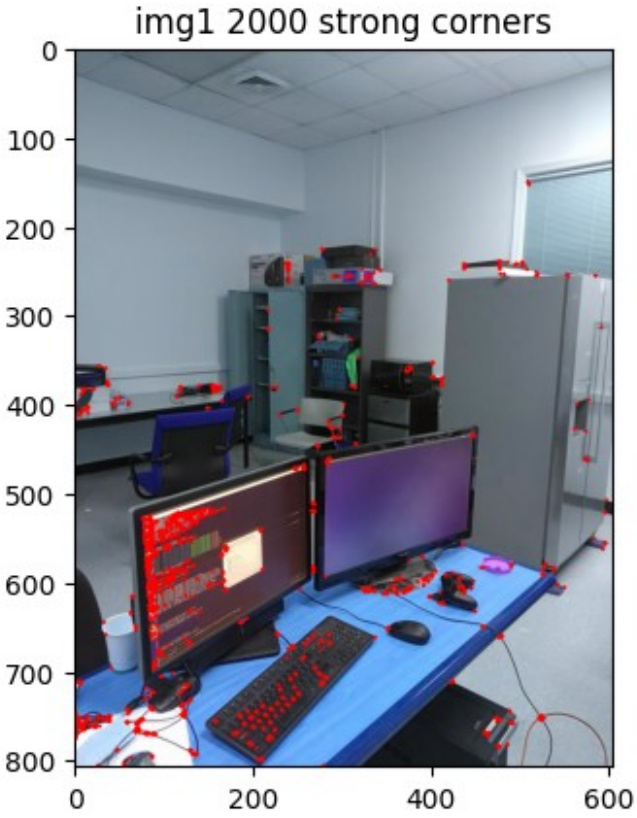

0
1000
2000
3000
4000
5000
6000
7000
8000
0
2000
4000

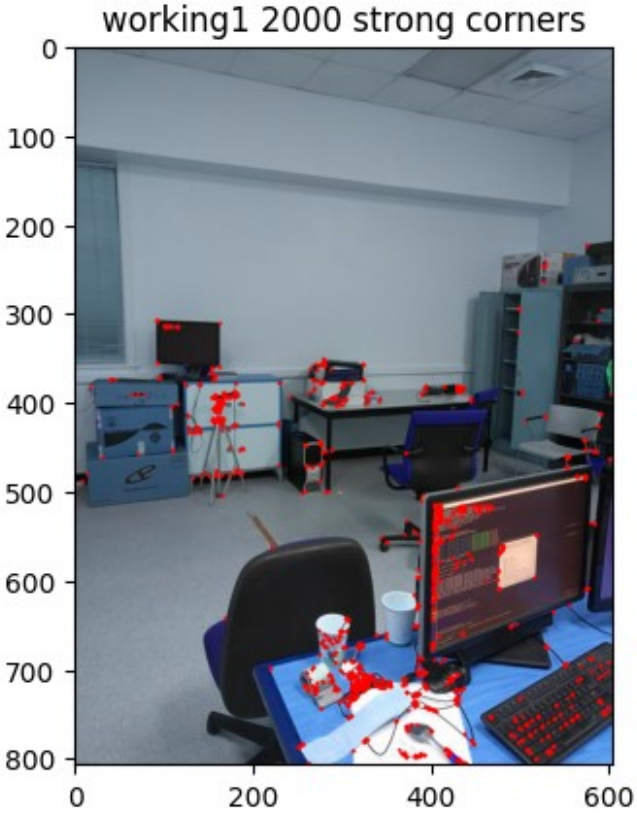
Set 6 [5pts] (Indoors)

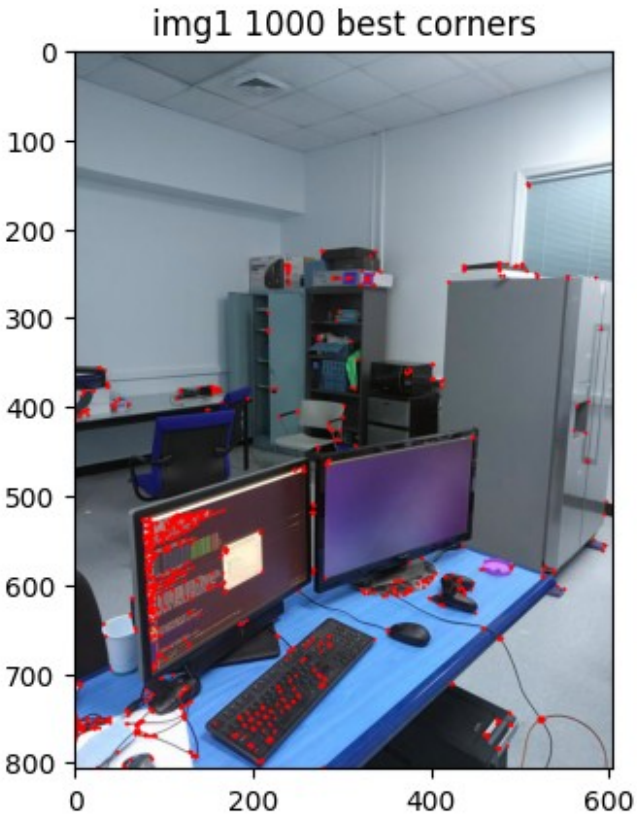
```
# Indoors
files = gdown.download_folder(id="1RfDSald_h2kwxRbcu8iq6ZPJ8TjOEGzK",
quiet=True, use_cookies=False)
files = ['/content/Indoors/1.jpg', '/content/Indoors/2.jpg',
'/content/Indoors/3.jpg', '/content/Indoors/4.jpg',
'/content/Indoors/5.jpg', '/content/Indoors/6.jpg',
'/content/Indoors/7.jpg', '/content/Indoors/8.jpg']

print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = True,  feature_descriptor_method="mine")
#img = mypano(files, num_best_corners = 5000, display = False,
feature_descriptor_method="sift", ransac_sample_size = 4,
match_ratio_threshold=0.5)

show_img(img)
```

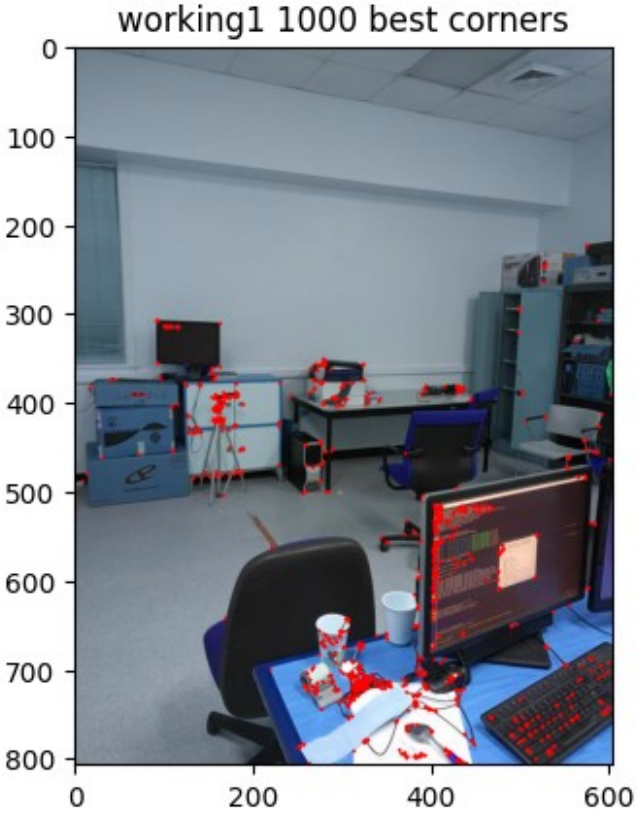
```
files: ['/content/Indoors/1.jpg', '/content/Indoors/2.jpg',
'/content/Indoors/3.jpg', '/content/Indoors/4.jpg',
'/content/Indoors/5.jpg', '/content/Indoors/6.jpg',
'/content/Indoors/7.jpg', '/content/Indoors/8.jpg']
```

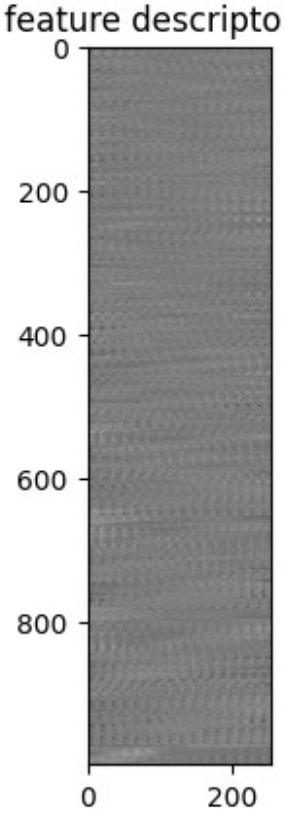


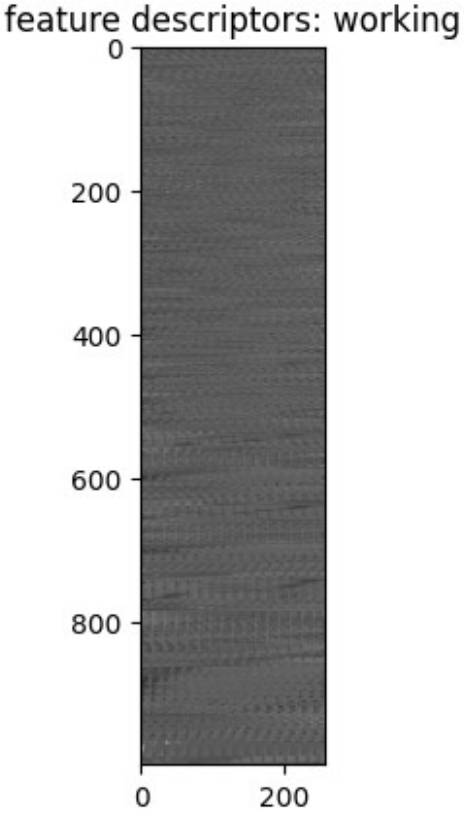

img1 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

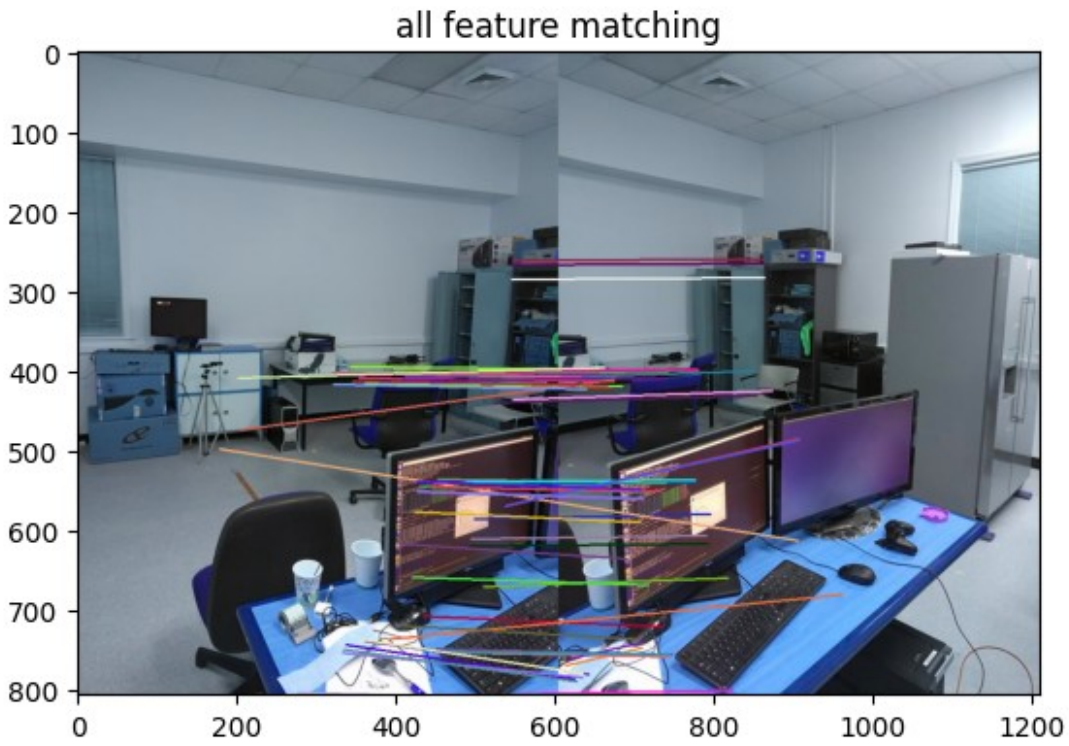

working1 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600


img1 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

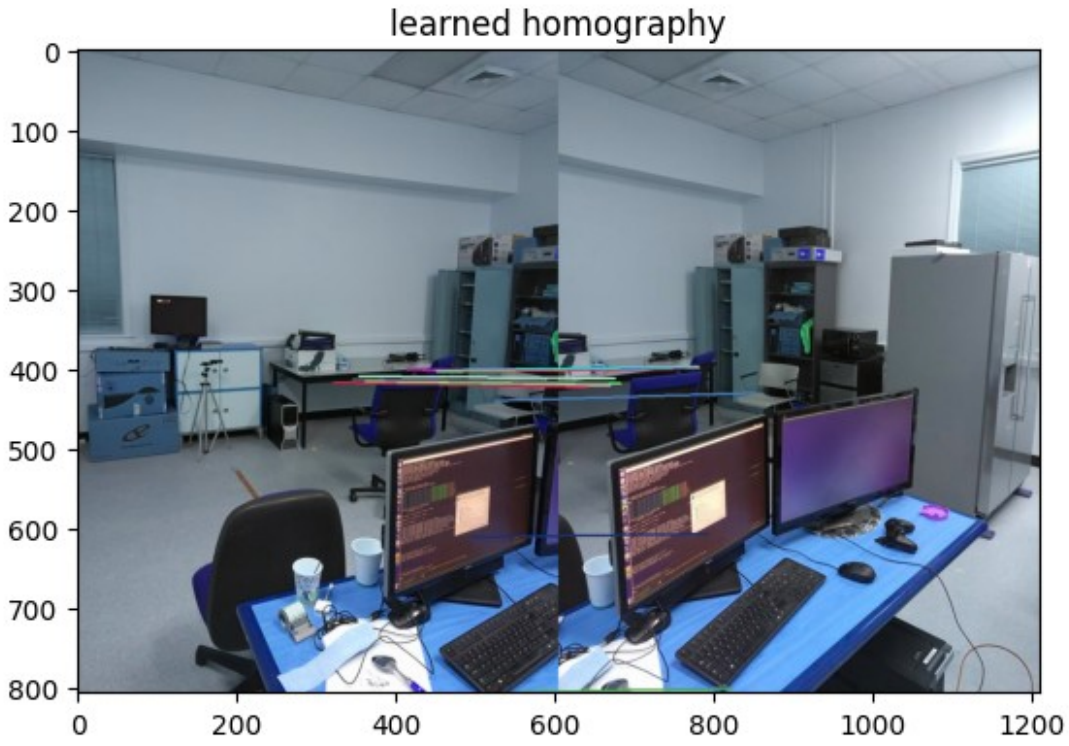

working1 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

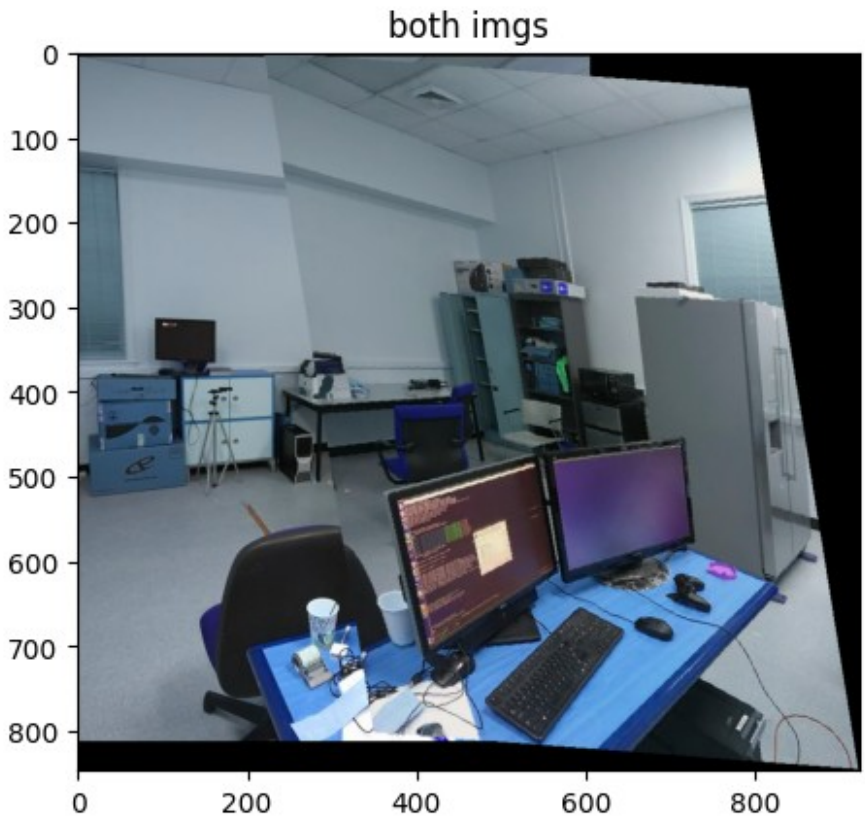

feature descriptors: 1
0
200
400
600
800
0
200

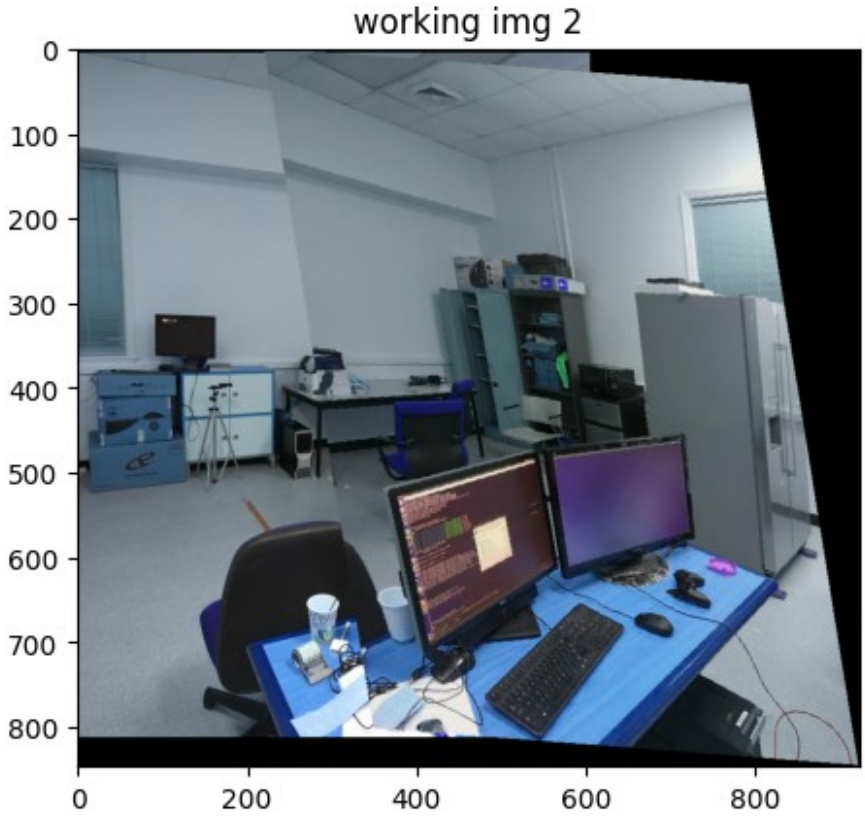

feature descriptors: working
0
200
400
600
800
0
200

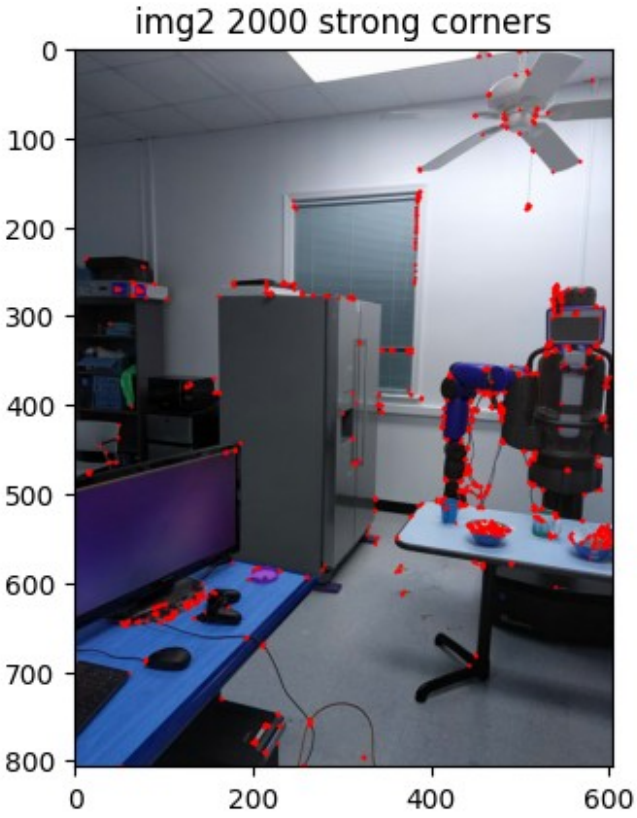


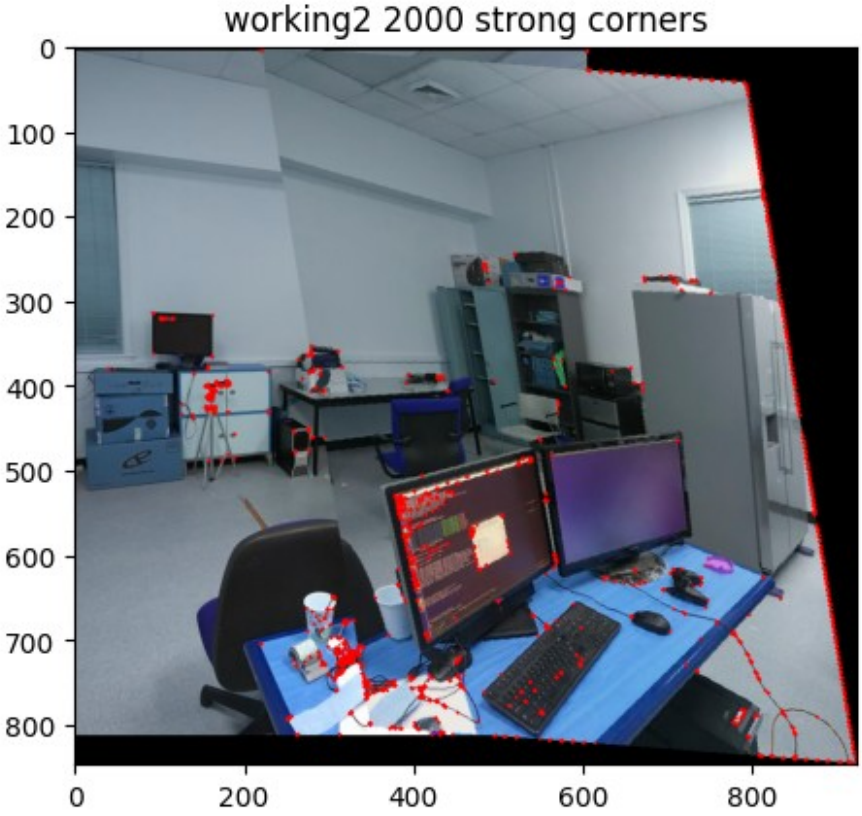

```
returning matrix with 12 inliers and mean error 47.250620412502194,
ranout of iterations
(12, 4)
```

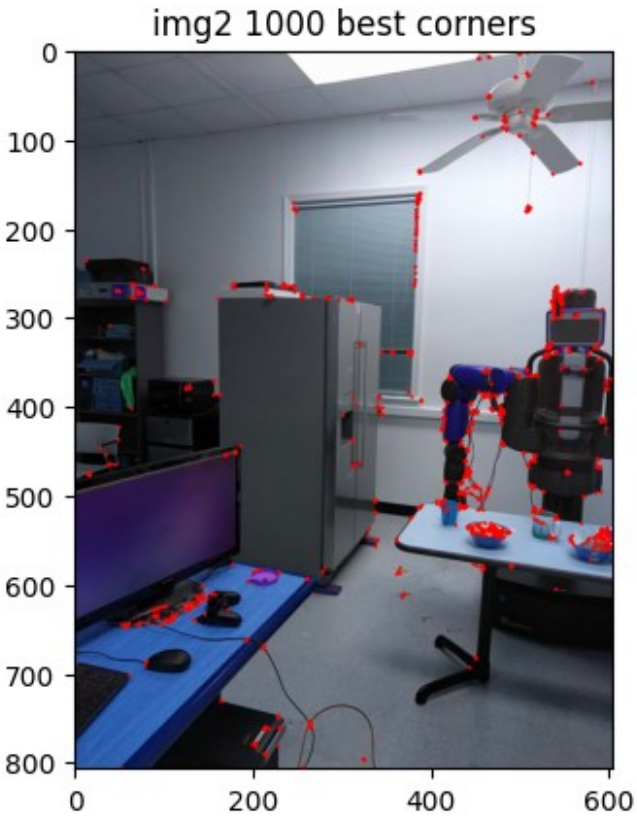


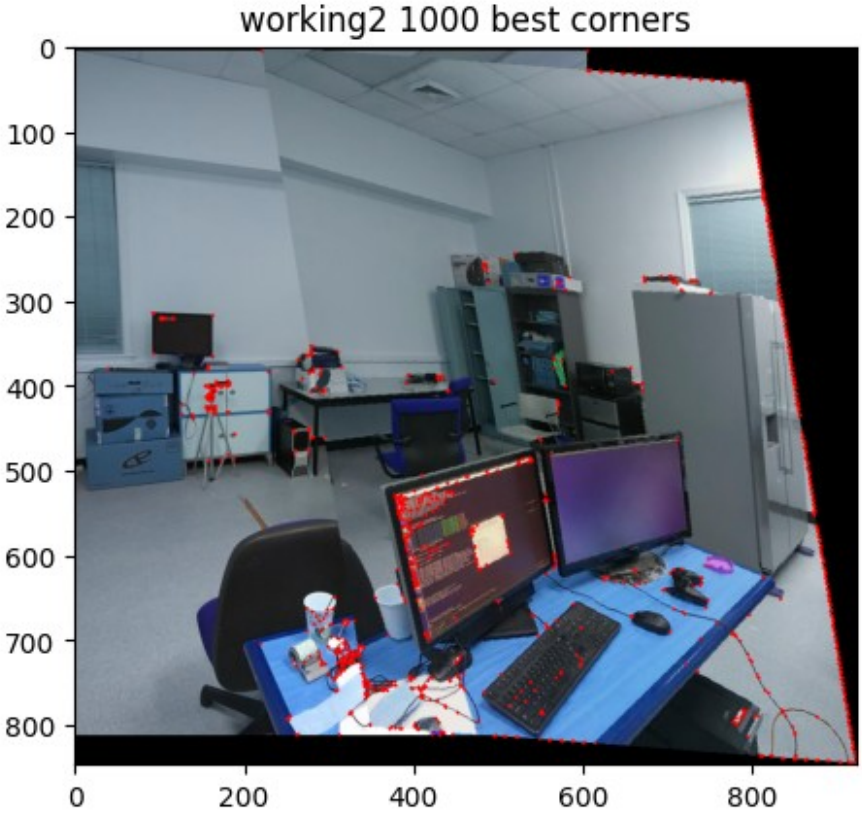

both imgs
0
100
200
300
400
500
600
700
800
0
200
400
600
800

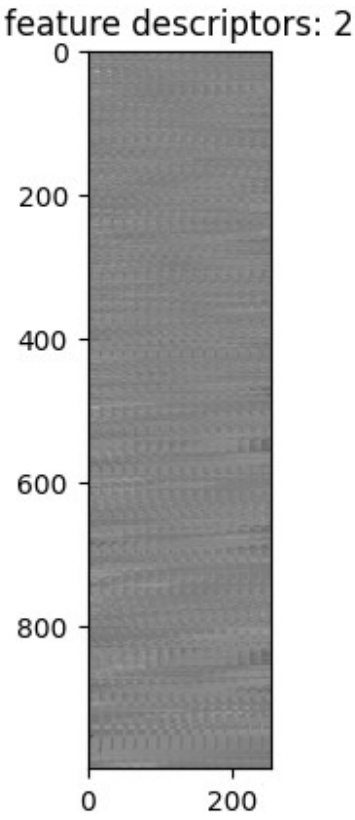

working img 2
0
100
200
300
400
500
600
700
800
0
200
400
600
800

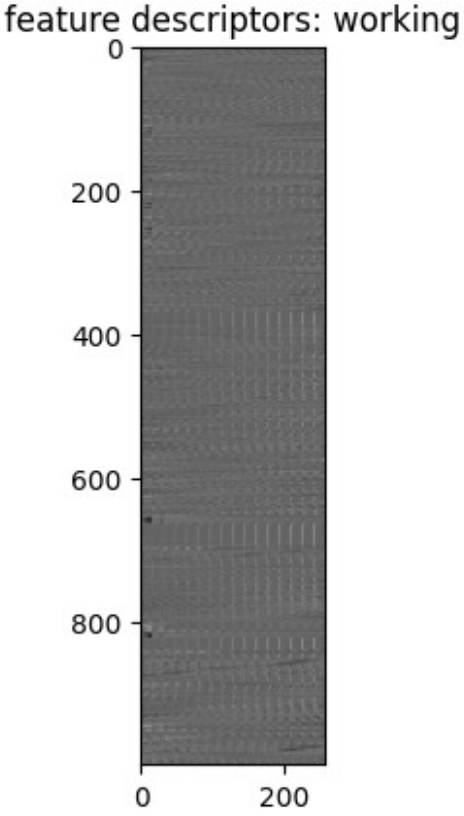

img2 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

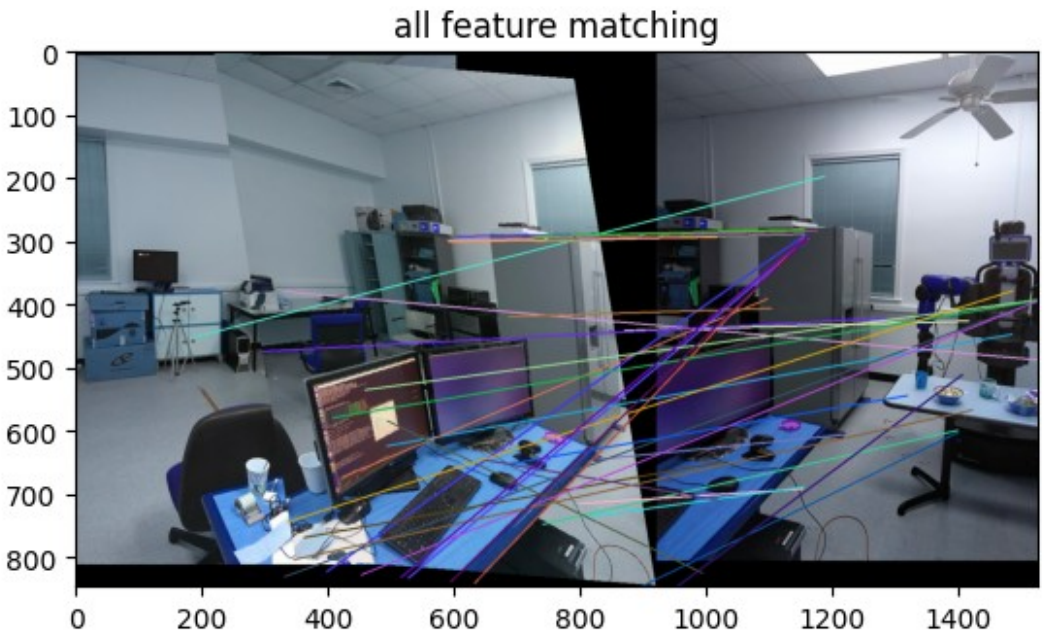

working2 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600
800


img2 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

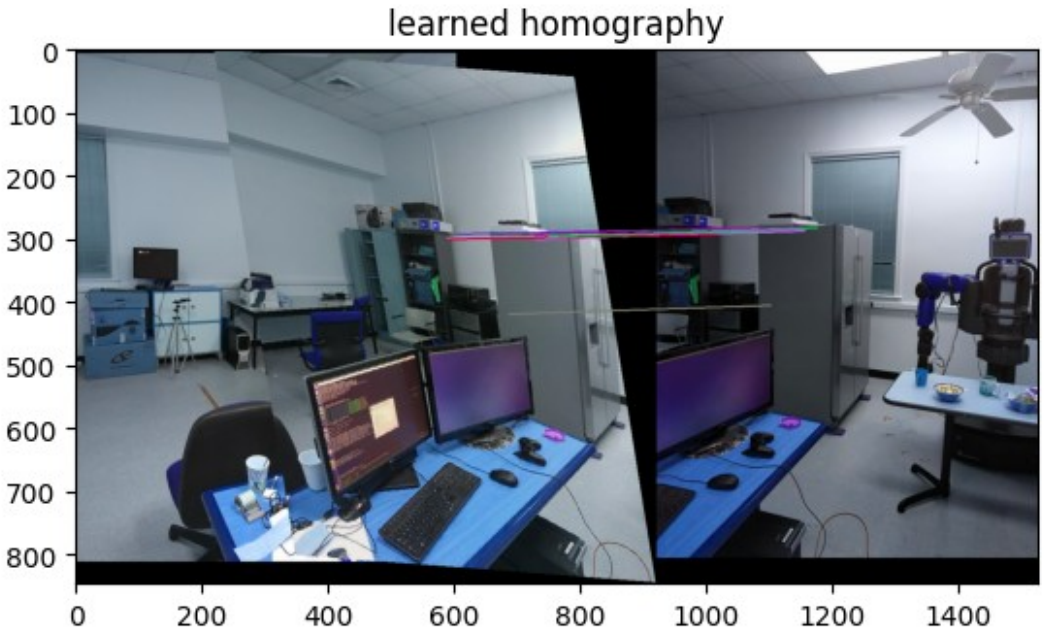

working2 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600
800

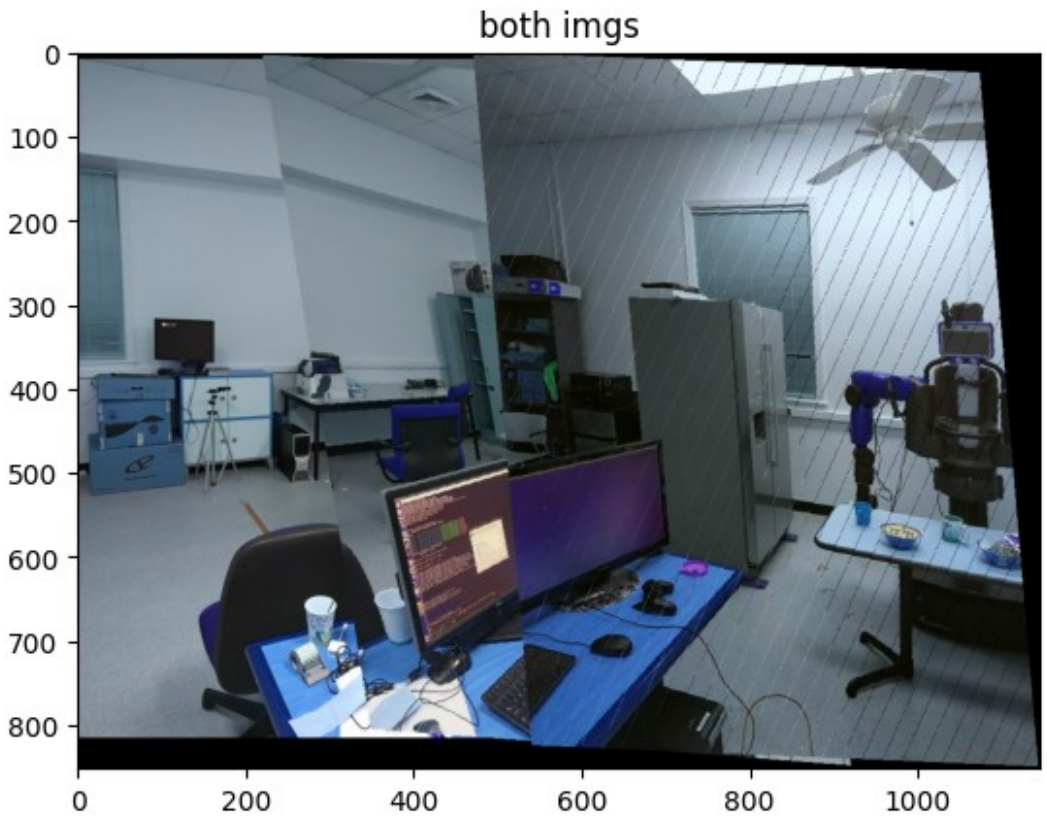

feature descriptors: 2
0
200
400
600
800
0
200

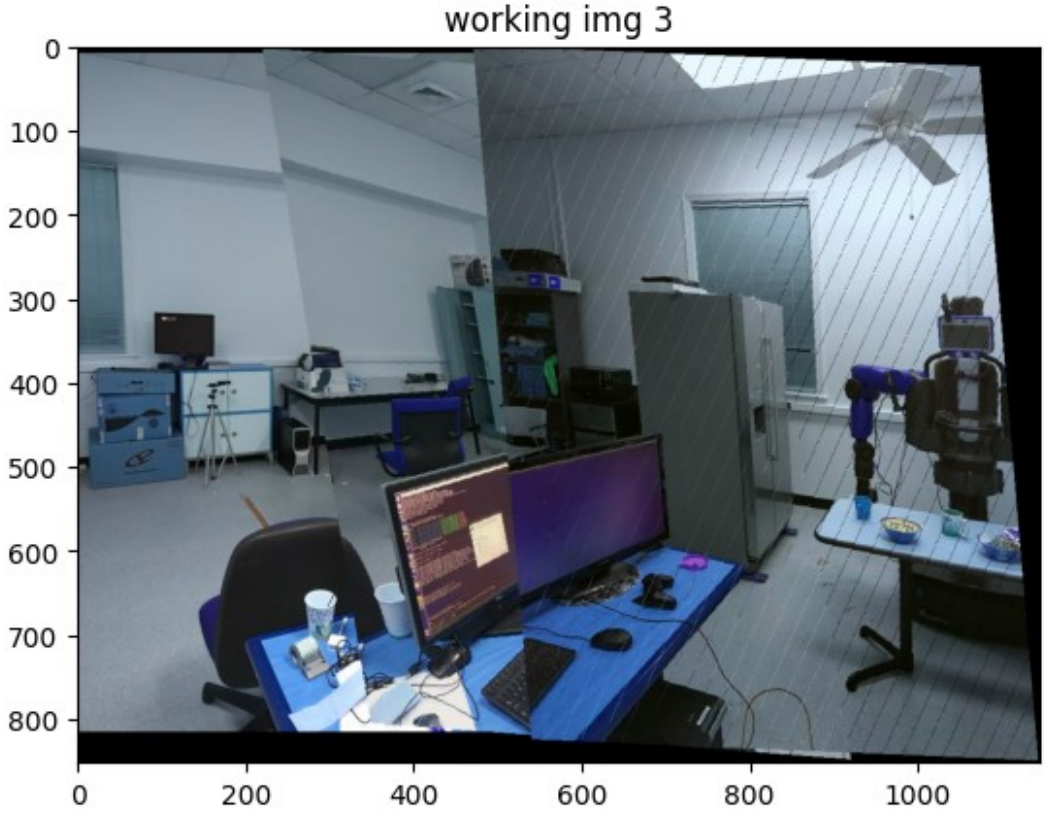

feature descriptors: working
0
200
400
600
800
0
200

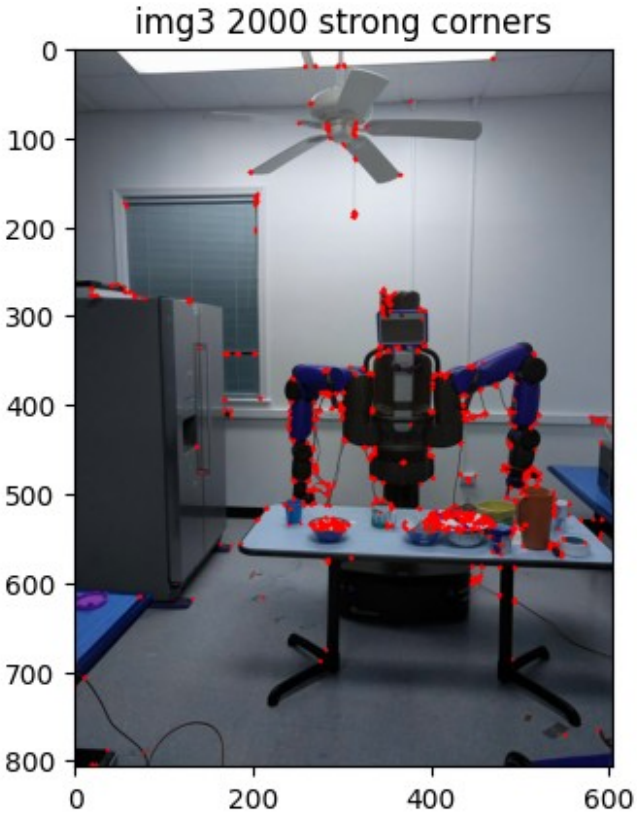


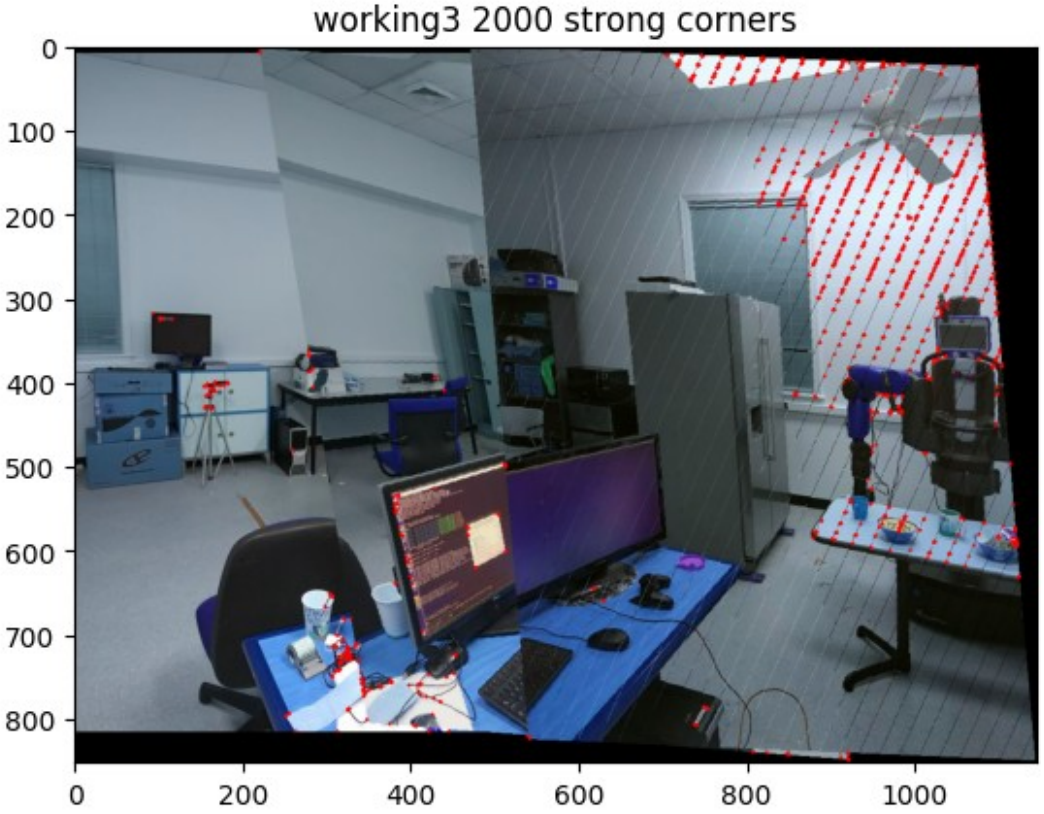

```
returning matrix with 9 inliers and mean error 345.95814113172503,
ranout of iterations
(9, 4)
```

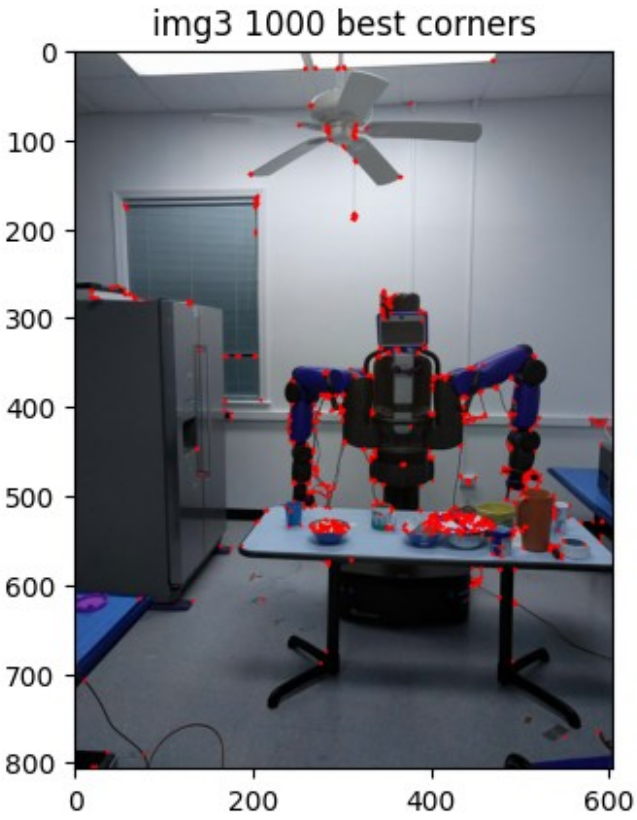


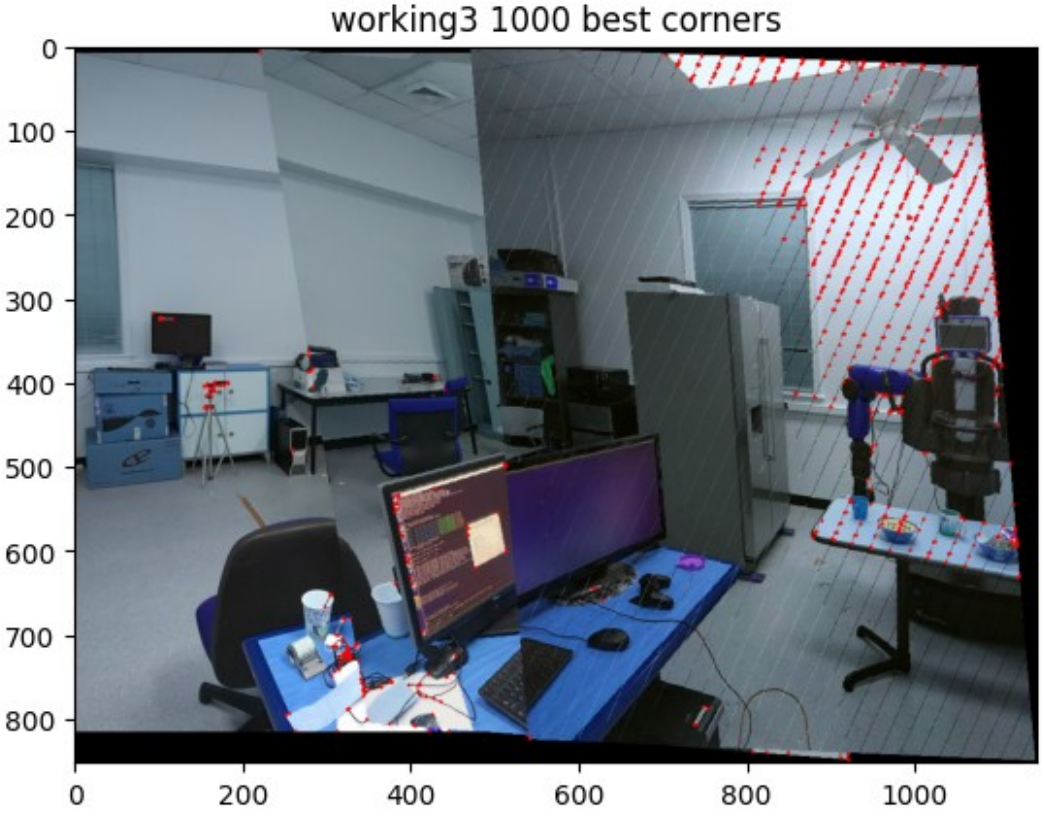

both imgs
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000

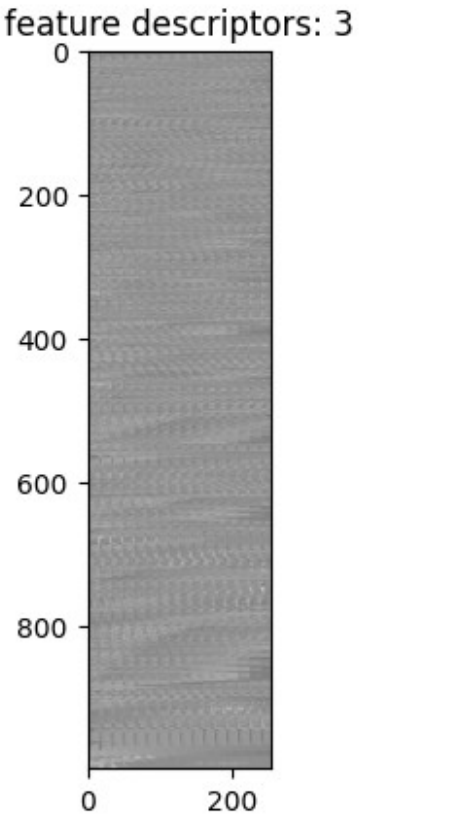

working img 3
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000

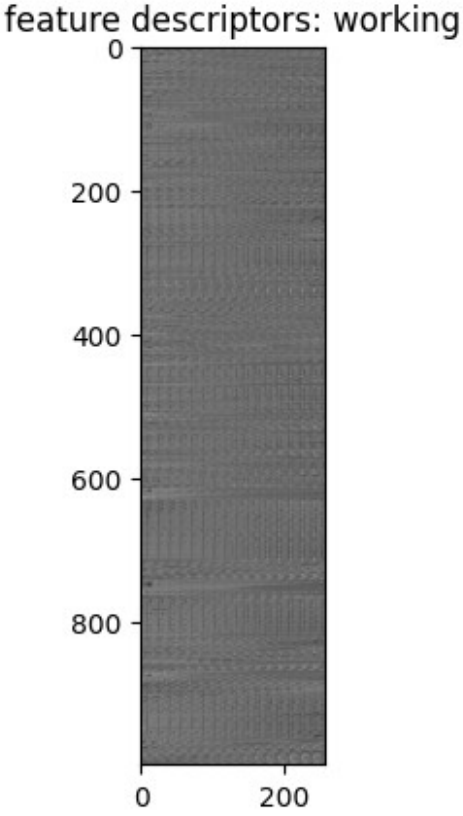

img3 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

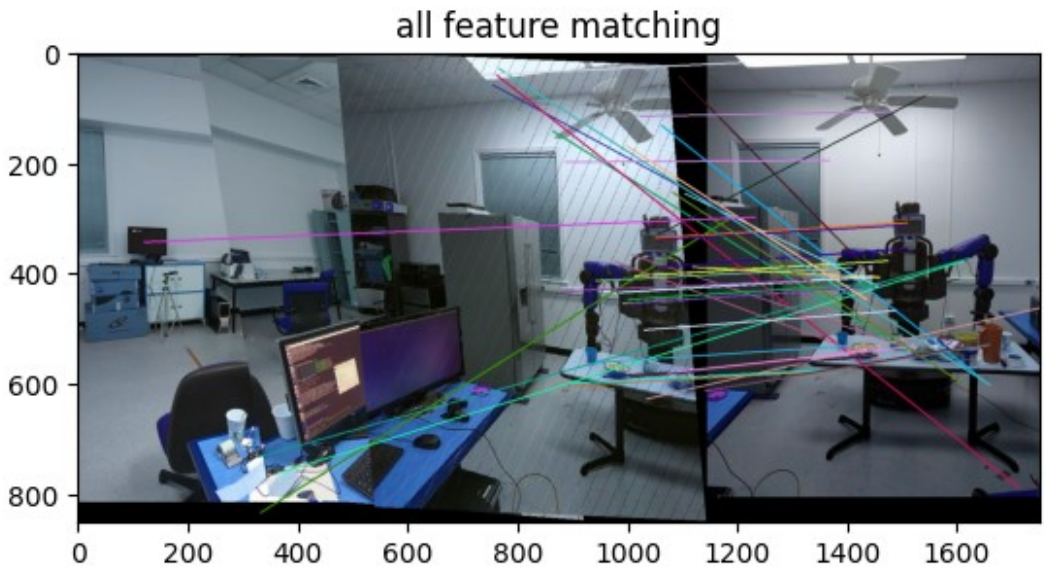

working3 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000


img3 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

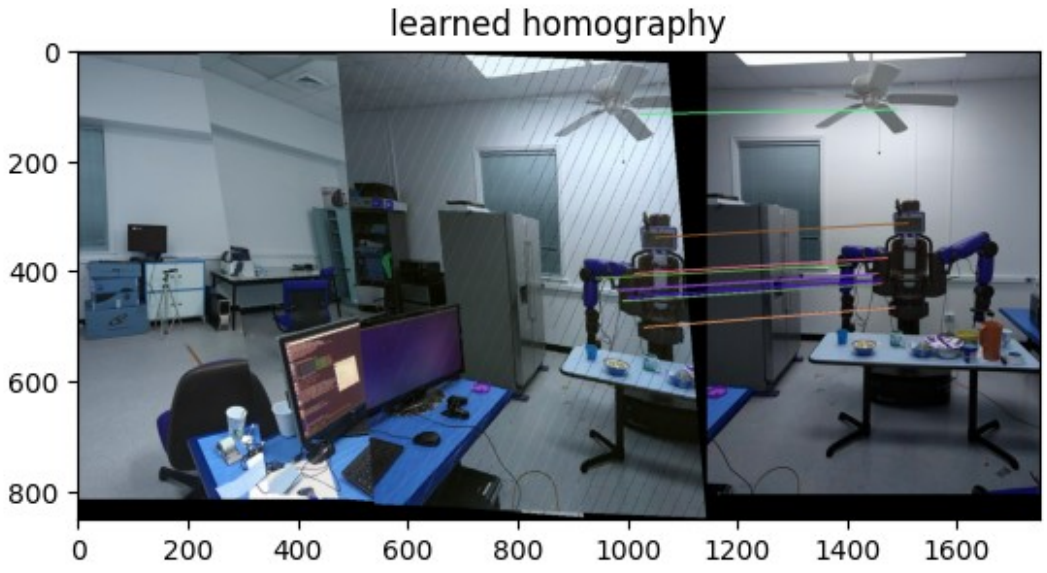

working3 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000

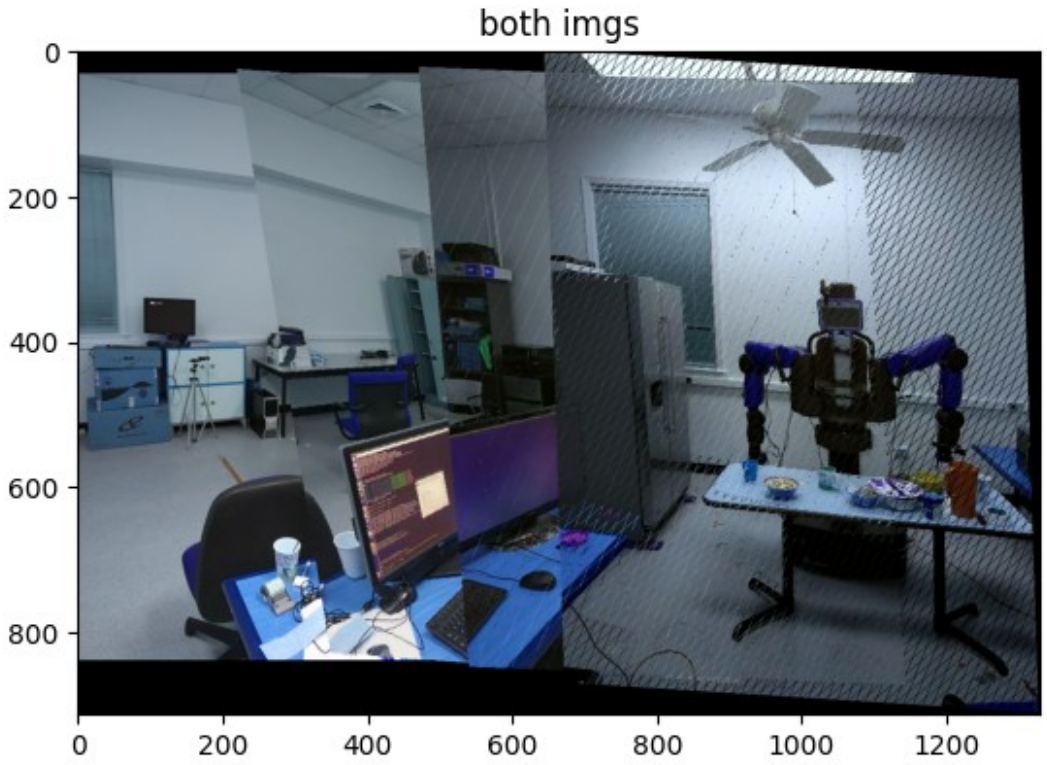

feature descriptors: 3
0
200
400
600
800
0
200

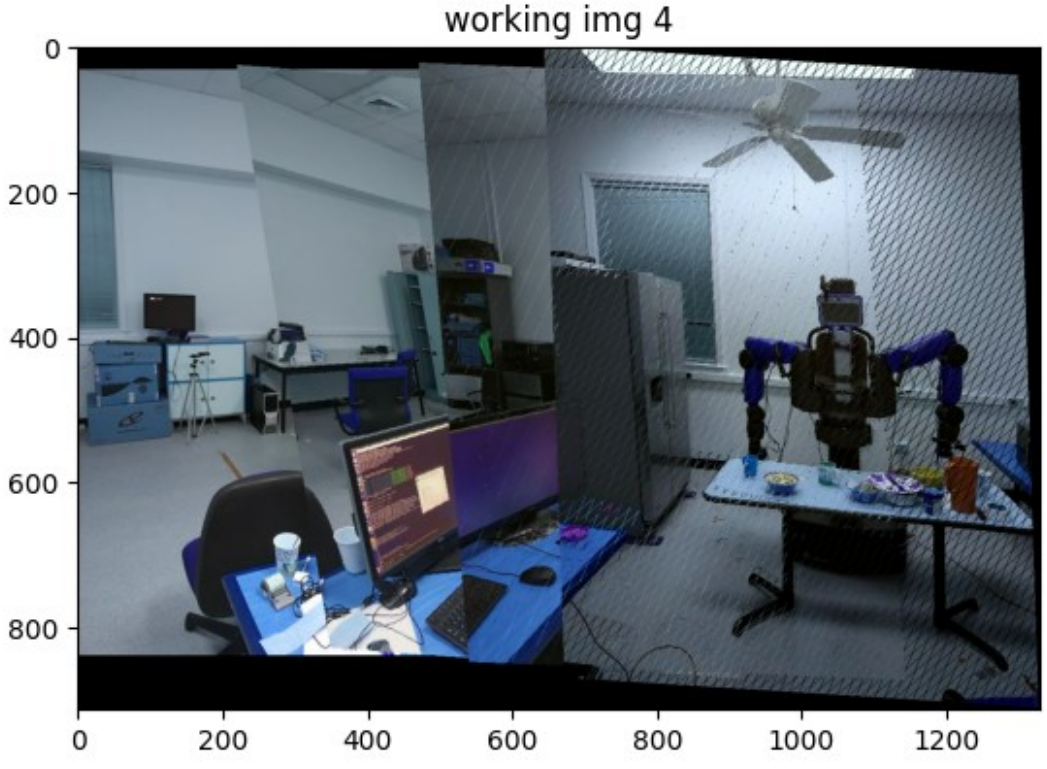

feature descriptors: working
0
200
400
600
800
0
200

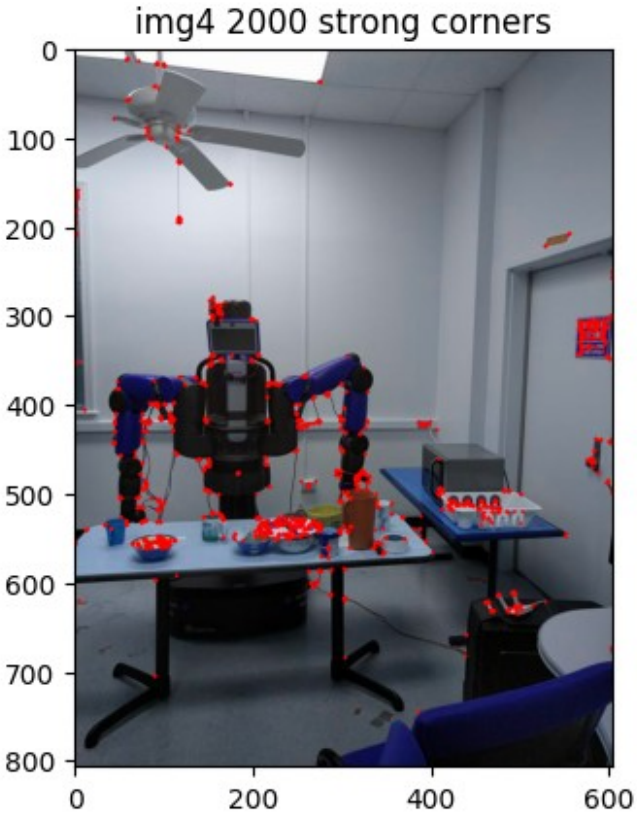


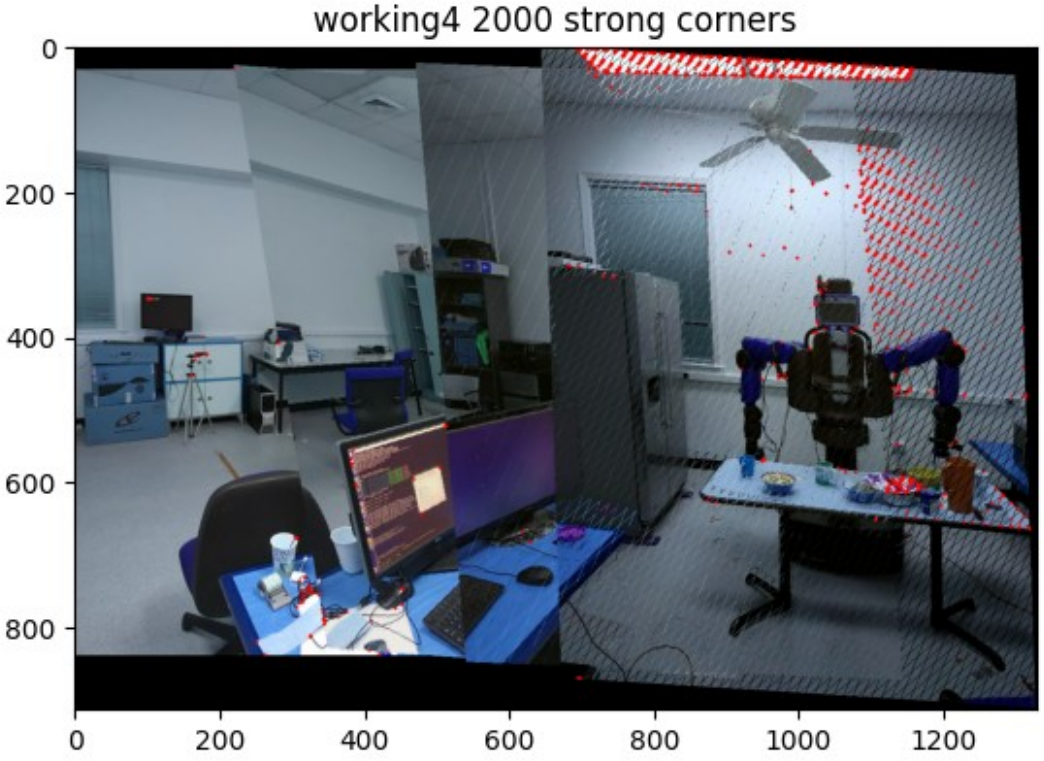

```
returning matrix with 13 inliers and mean error 184.33324559218588,
ranout of iterations
(13, 4)
```

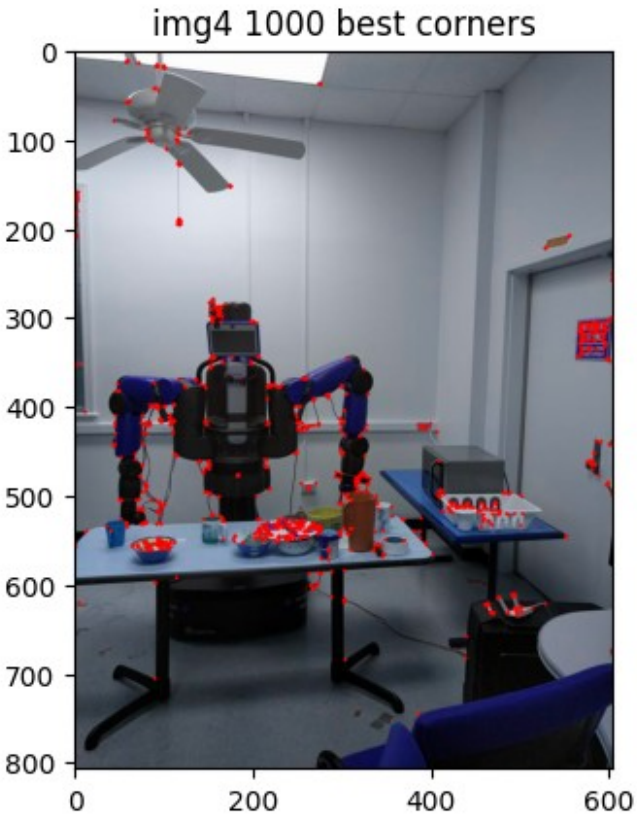


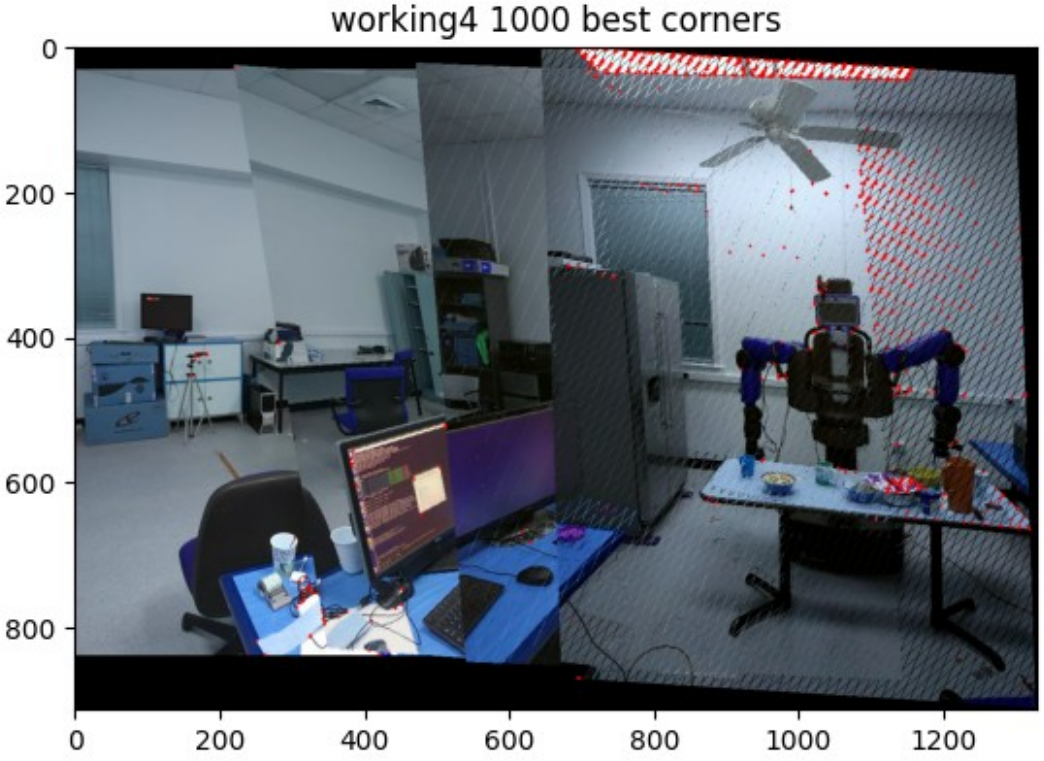

both imgs
0
200
400
600
800
0
200
400
600
800
1000
1200

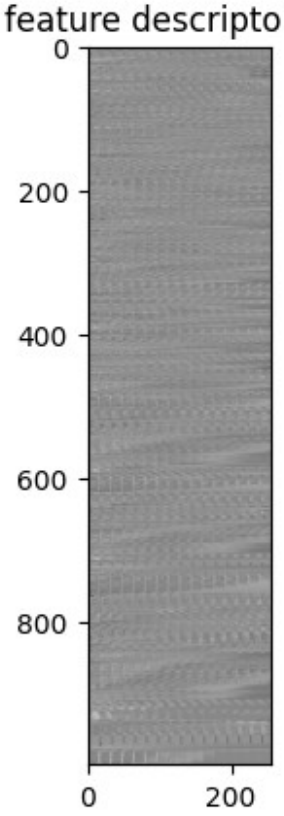

working img 4
0
200
400
600
800
0
200
400
600
800
1000
1200

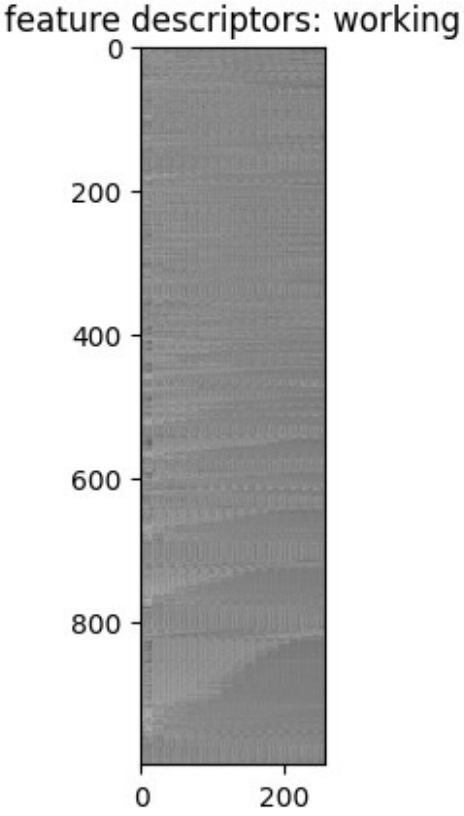

img4 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

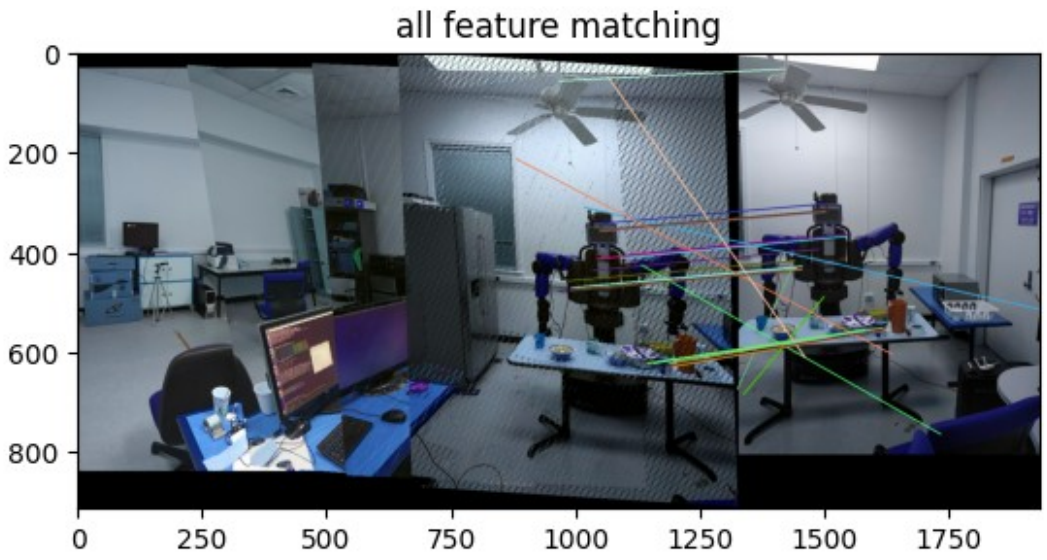

working4 2000 strong corners
0
200
400
600
800
0
200
400
600
800
1000
1200


img4 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

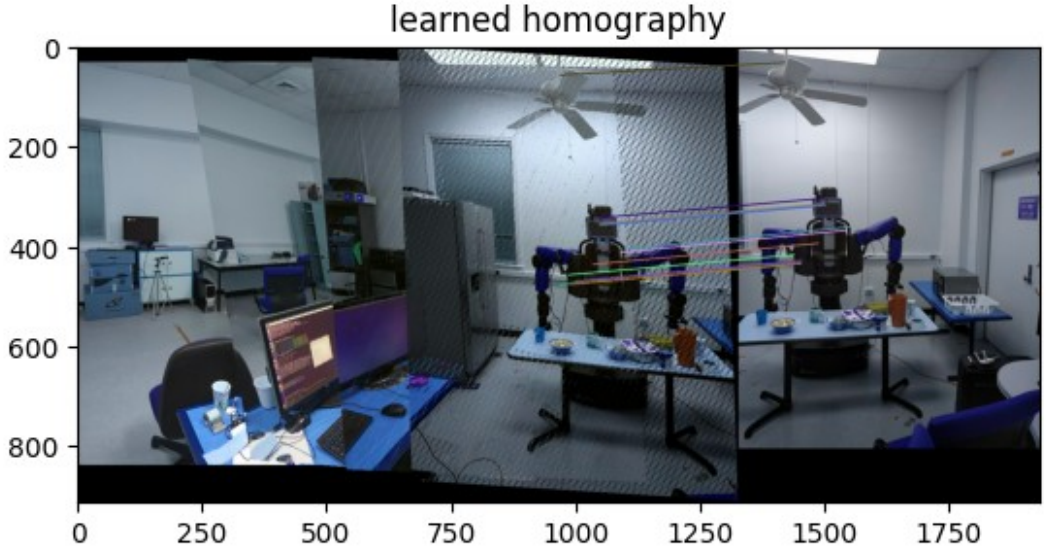

working4 1000 best corners
0
200
400
600
800
0
200
400
600
800
1000
1200

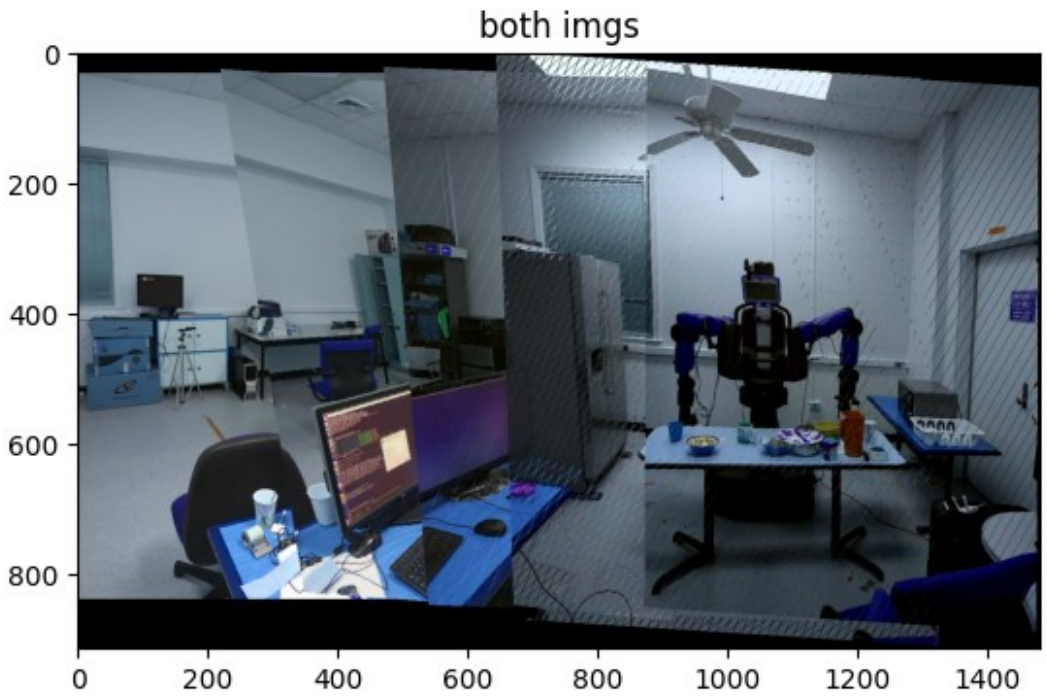

feature descriptors: 4
0
200
400
600
800
0
200

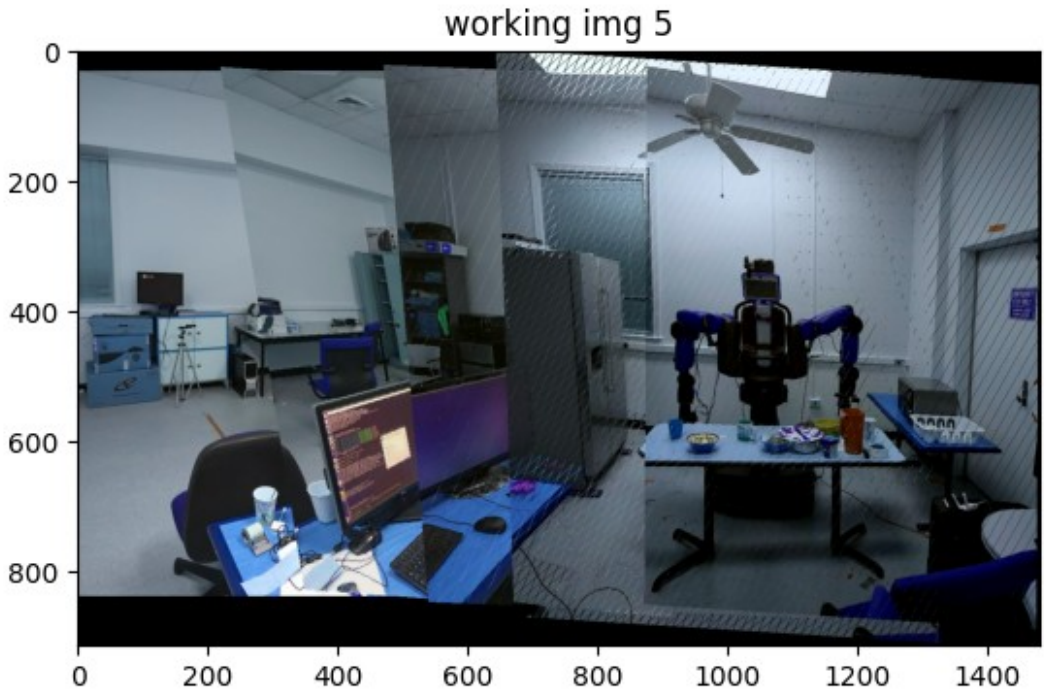

feature descriptors: working
0
200
400
600
800
0
200

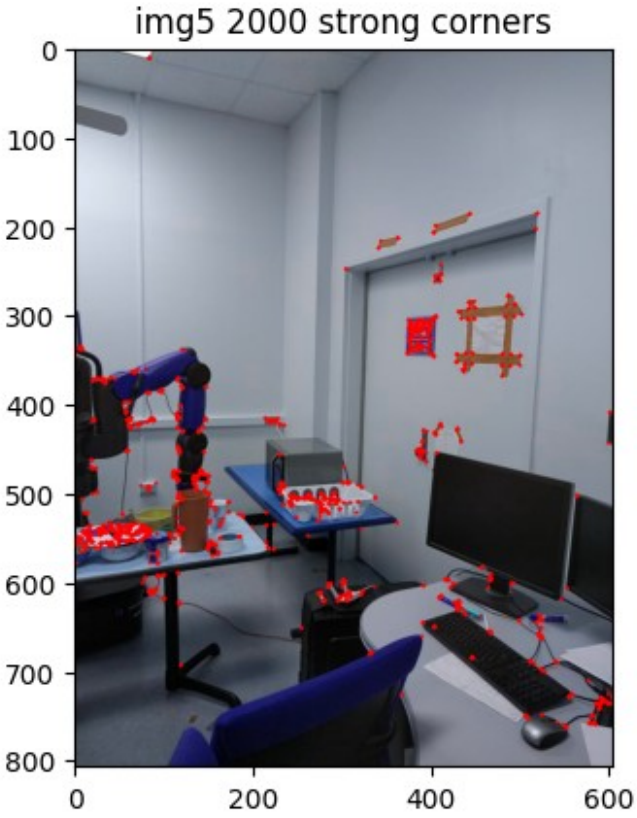


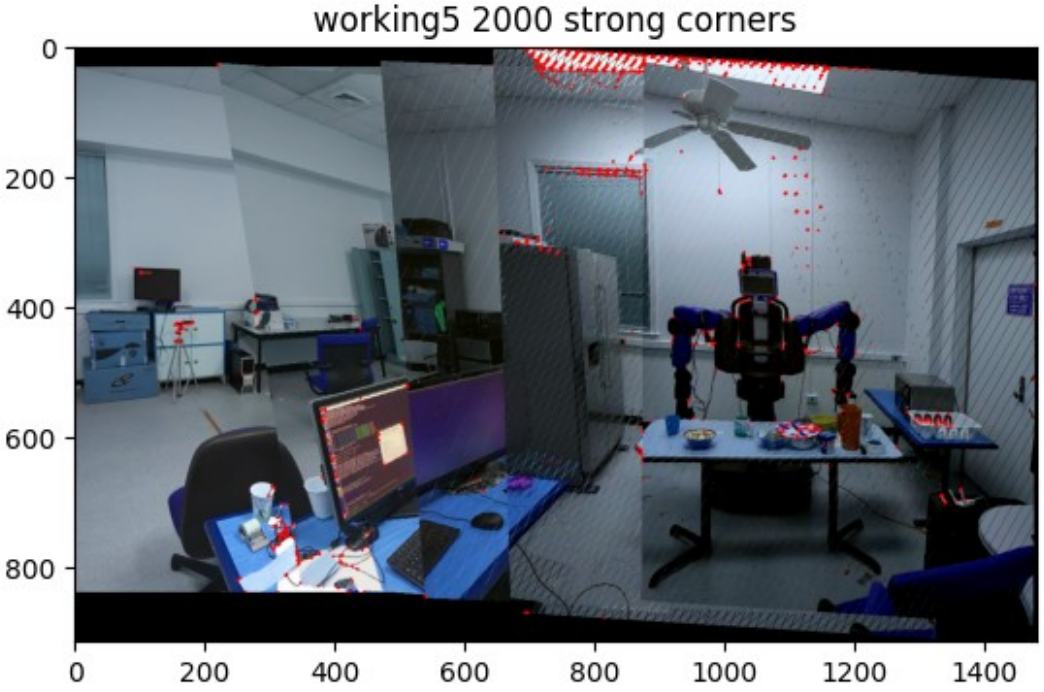

```
returning matrix with 13 inliers and mean error 116.70522804404392,
ranout of iterations
(13, 4)
```

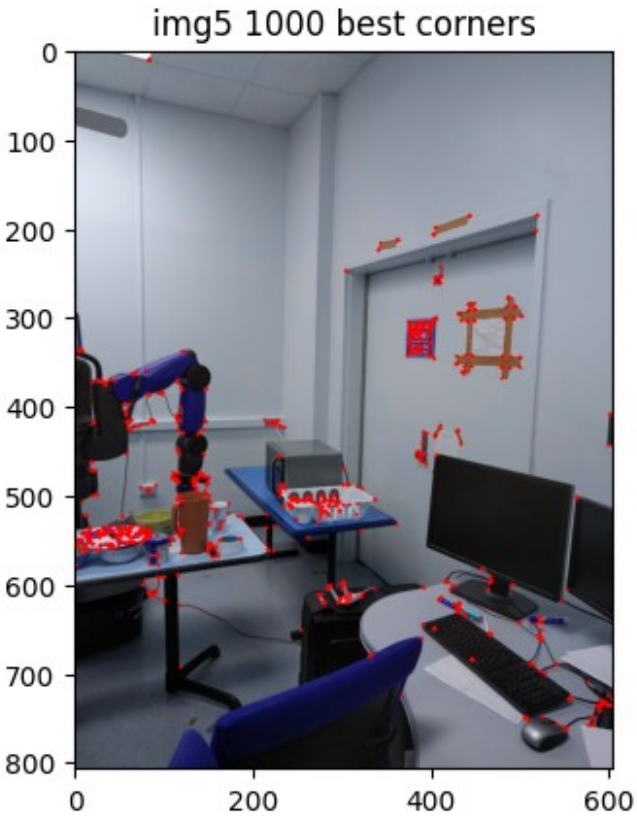


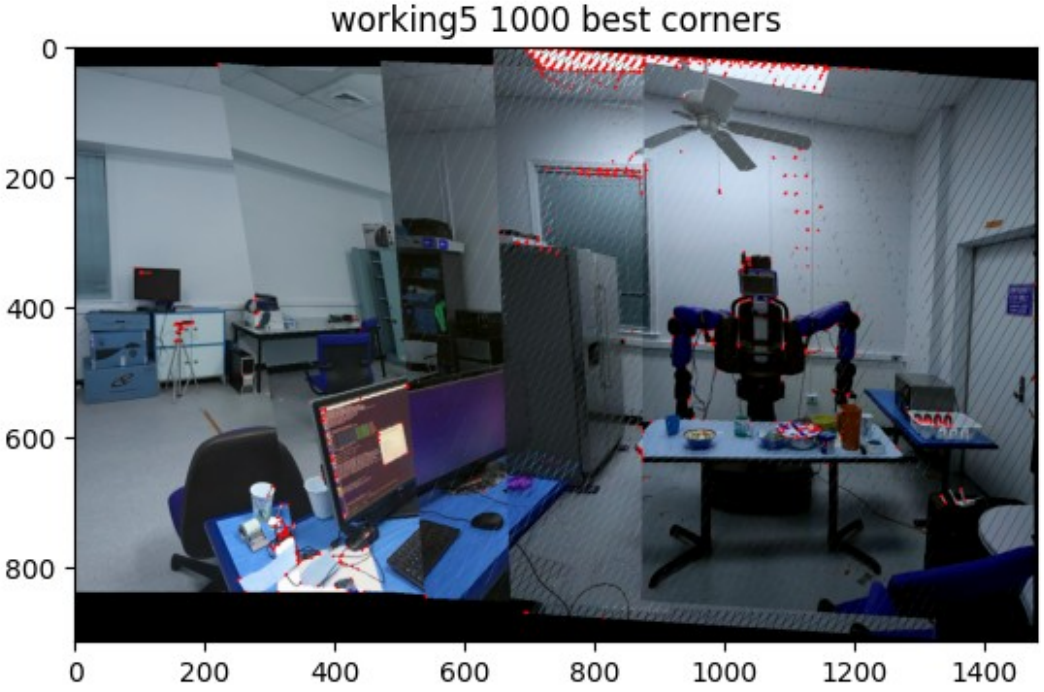

both imgs
0
200
400
600
800
0
200
400
600
800
1000
1200
1400

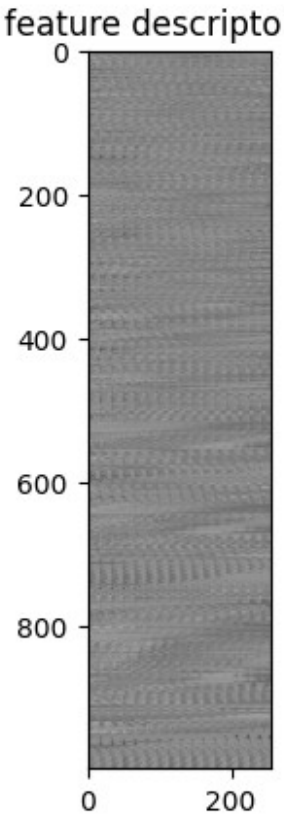

working img 5
0
200
400
600
800
0
200
400
600
800
1000
1200
1400

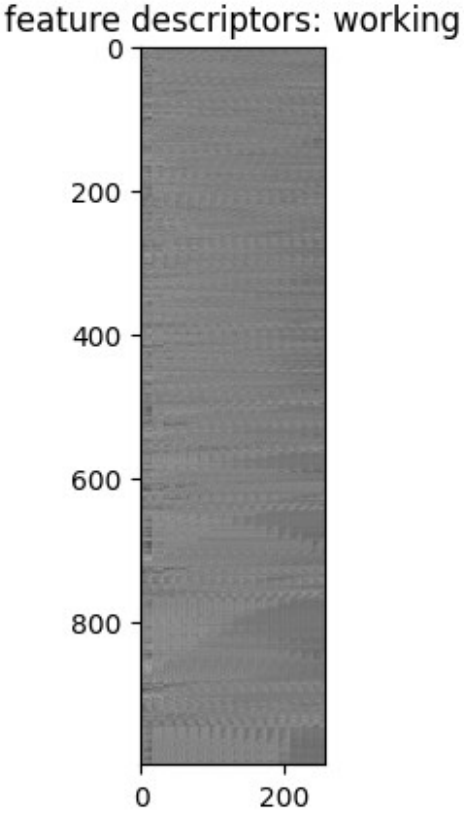

img5 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

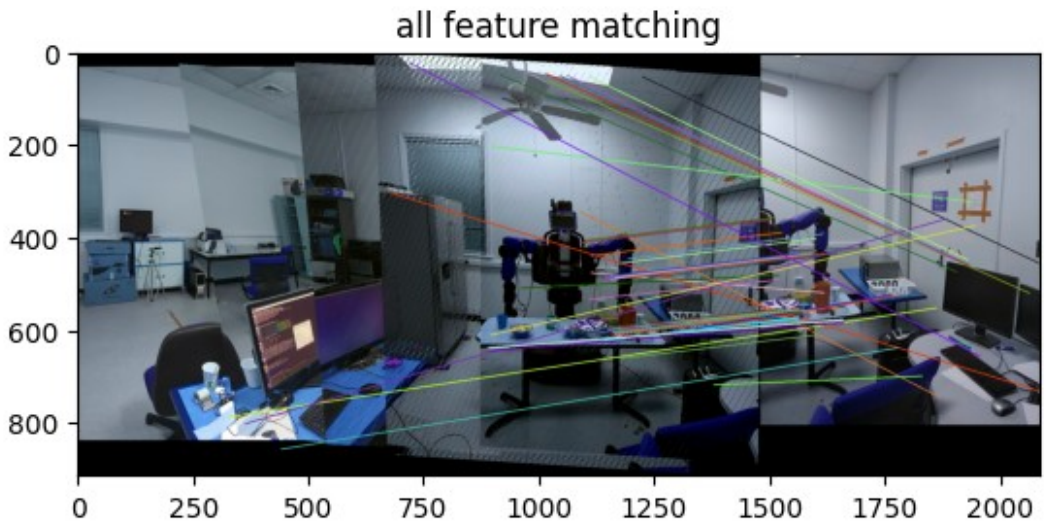

working5 2000 strong corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400


img5 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

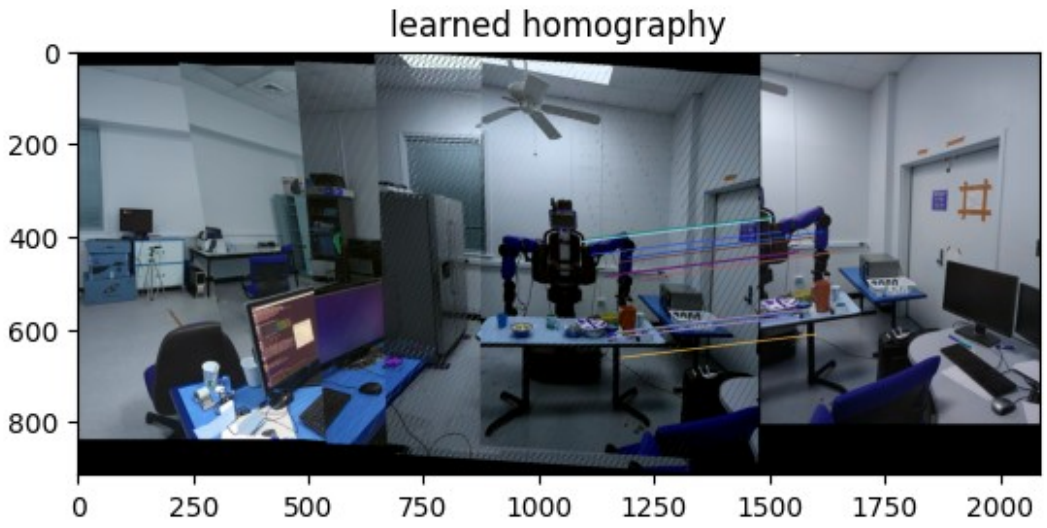

working5 1000 best corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400

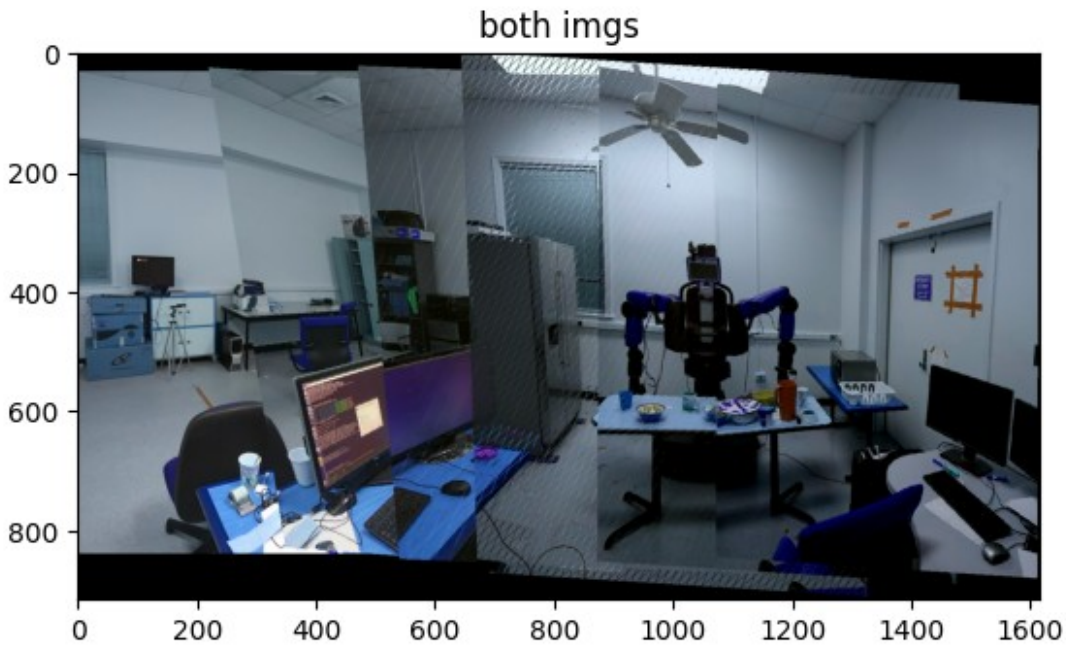

feature descriptors: 5
0
200
400
600
800
0
200

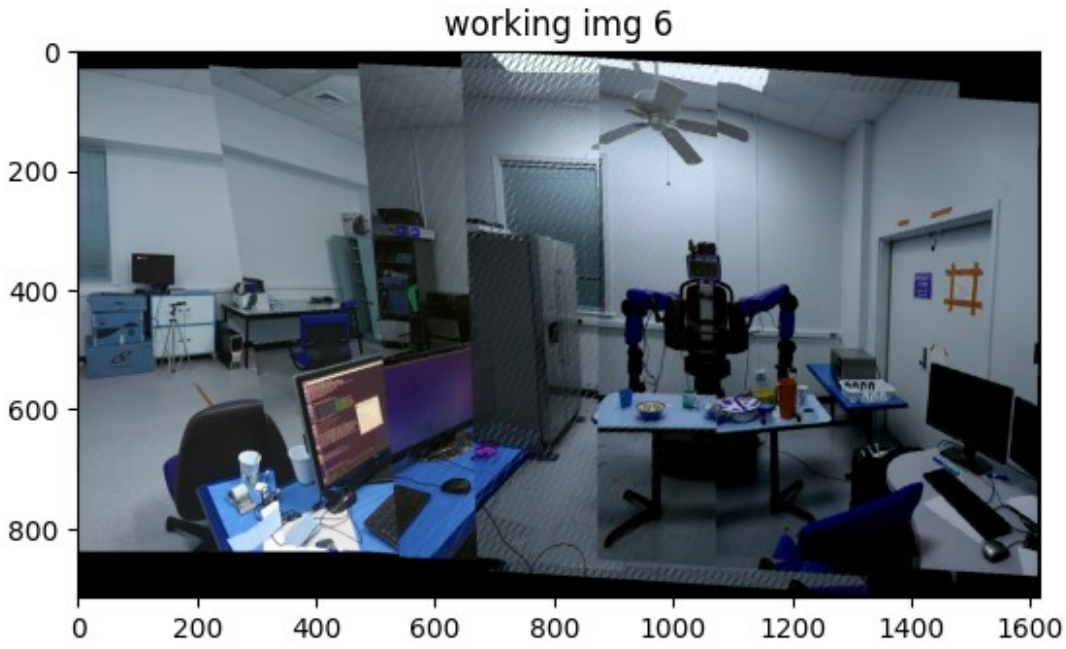

feature descriptors: working
0
200
400
600
800
0
200

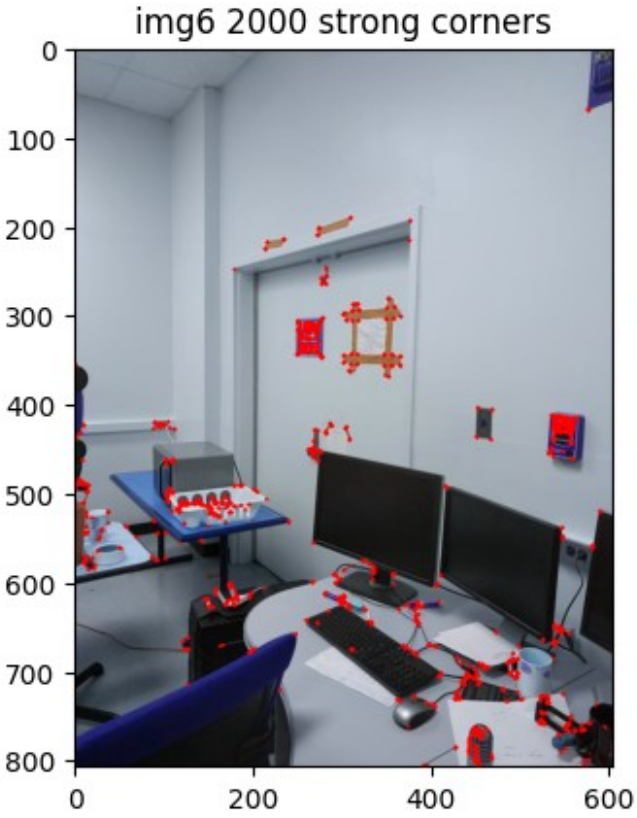


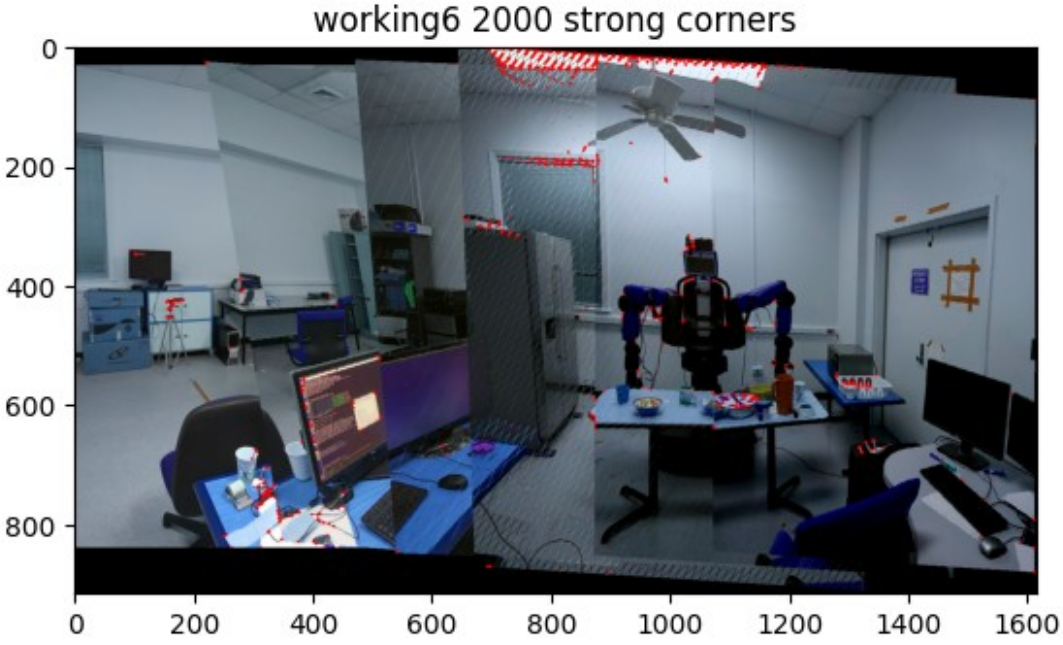

```
returning matrix with 14 inliers and mean error 296.1665493413947,
ranout of iterations
(14, 4)
```

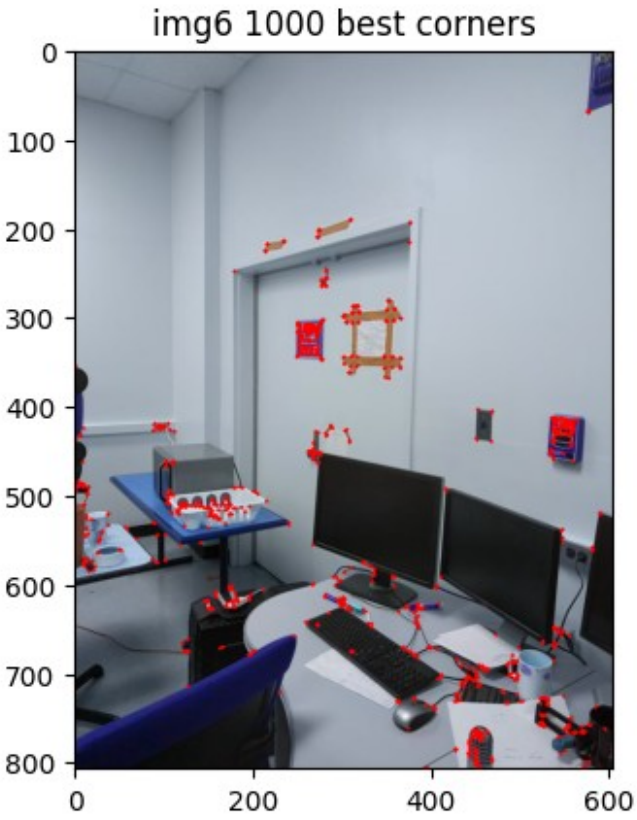


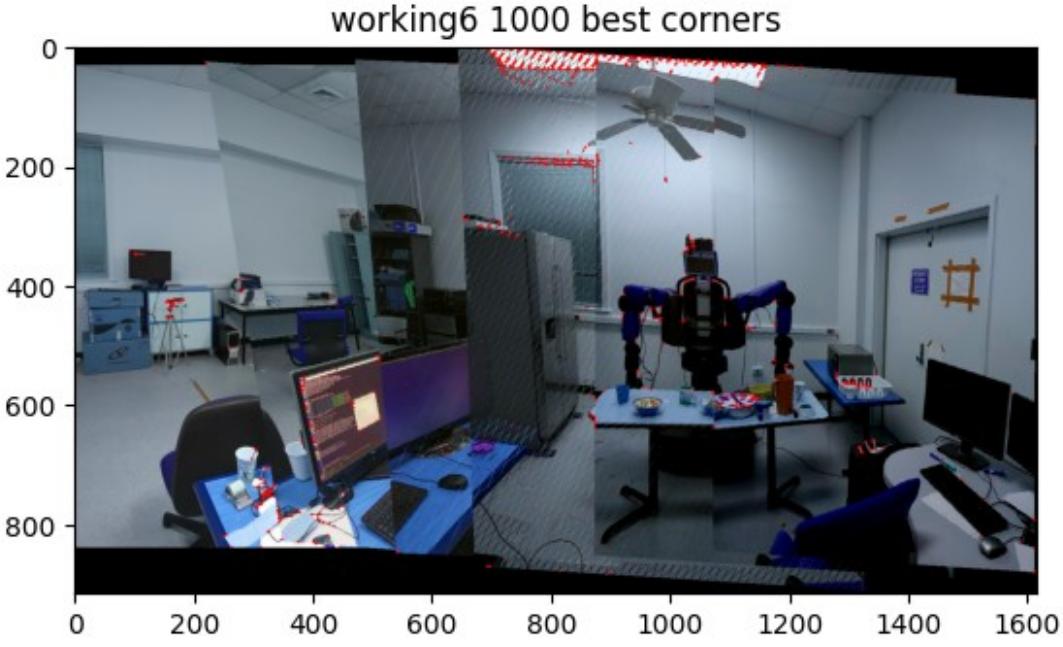

both imgs
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

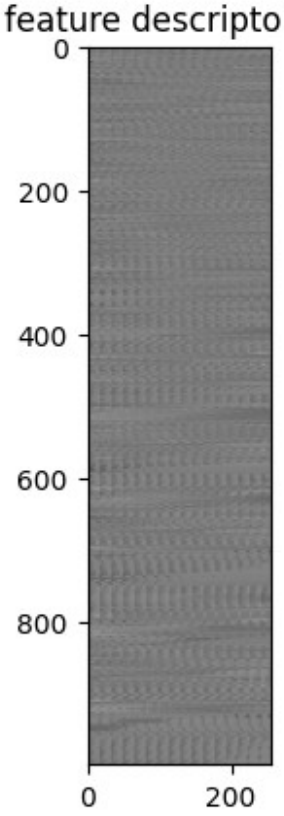

working img 6
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

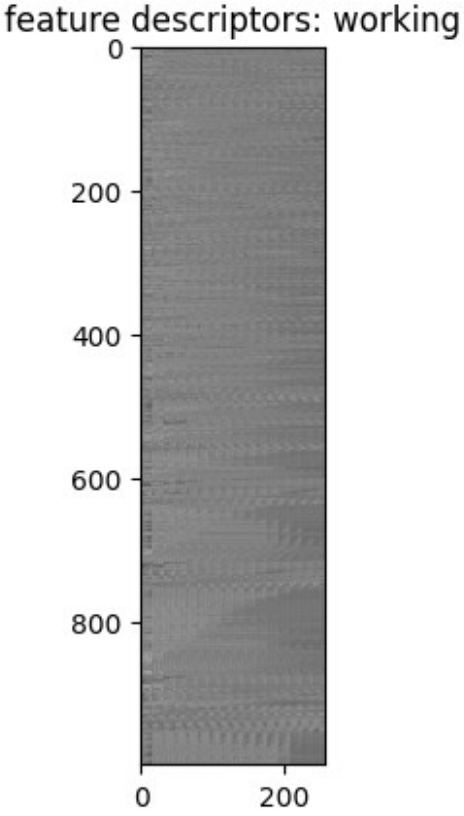

img6 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

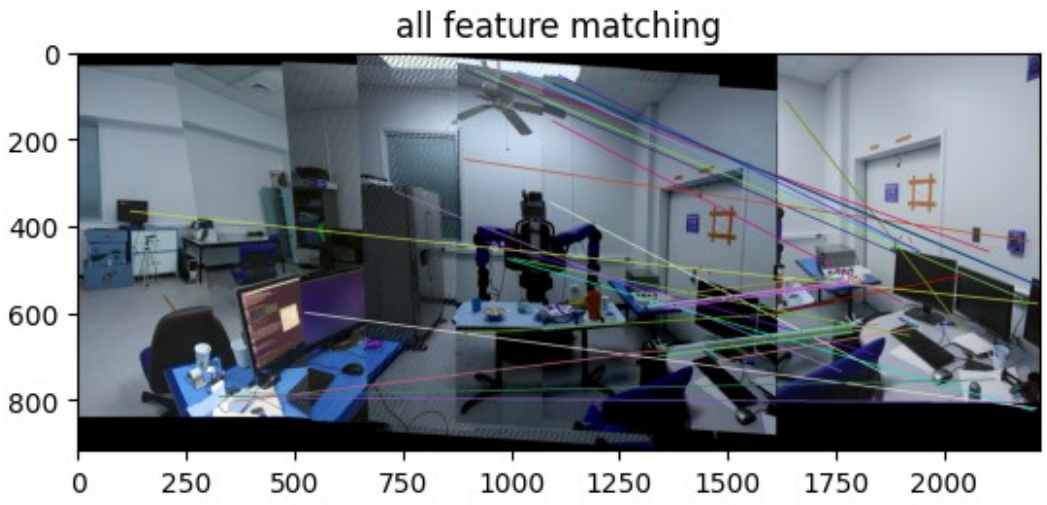

working6 2000 strong corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600


img6 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

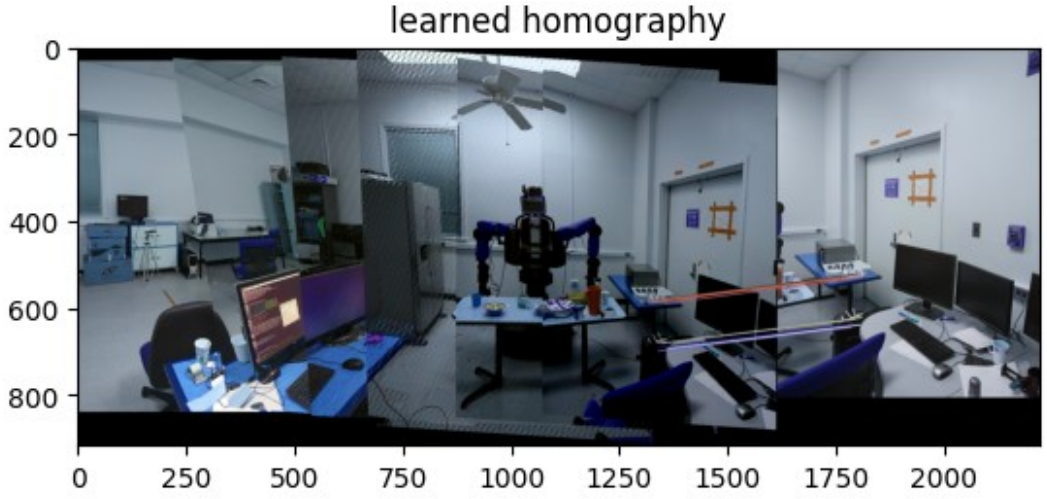

working6 1000 best corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

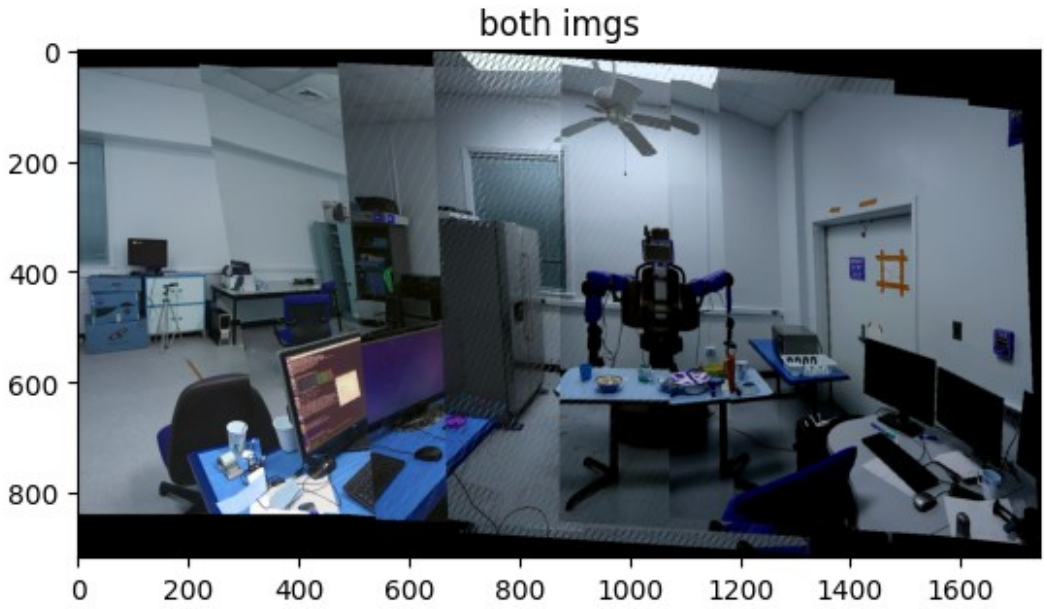

feature descriptors: 6
0
200
400
600
800
0
200

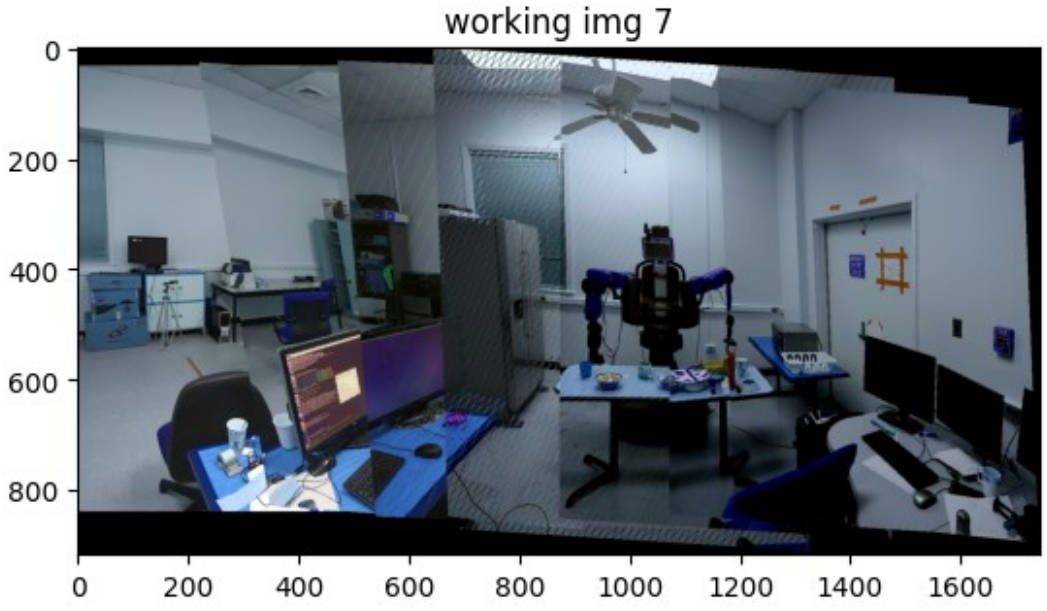

feature descriptors: working
0
200
400
600
800
0
200

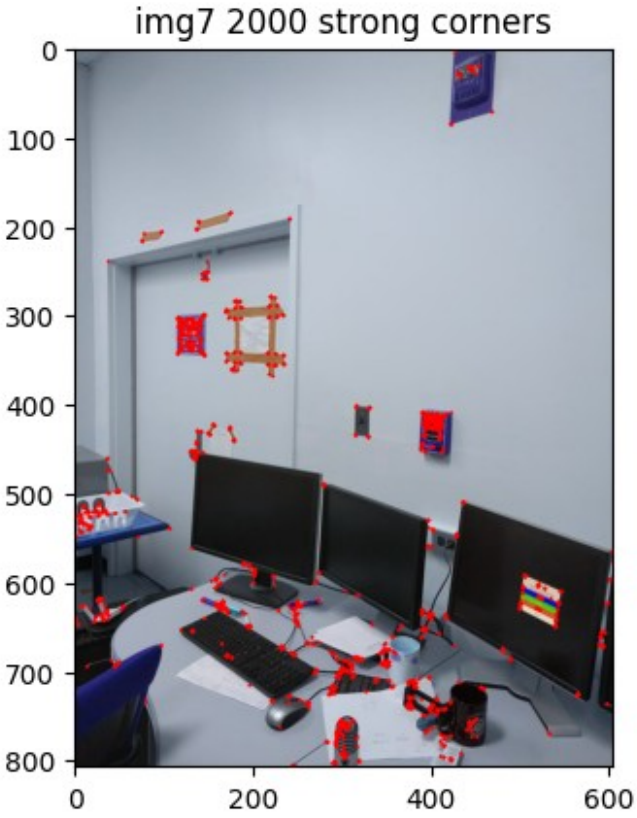


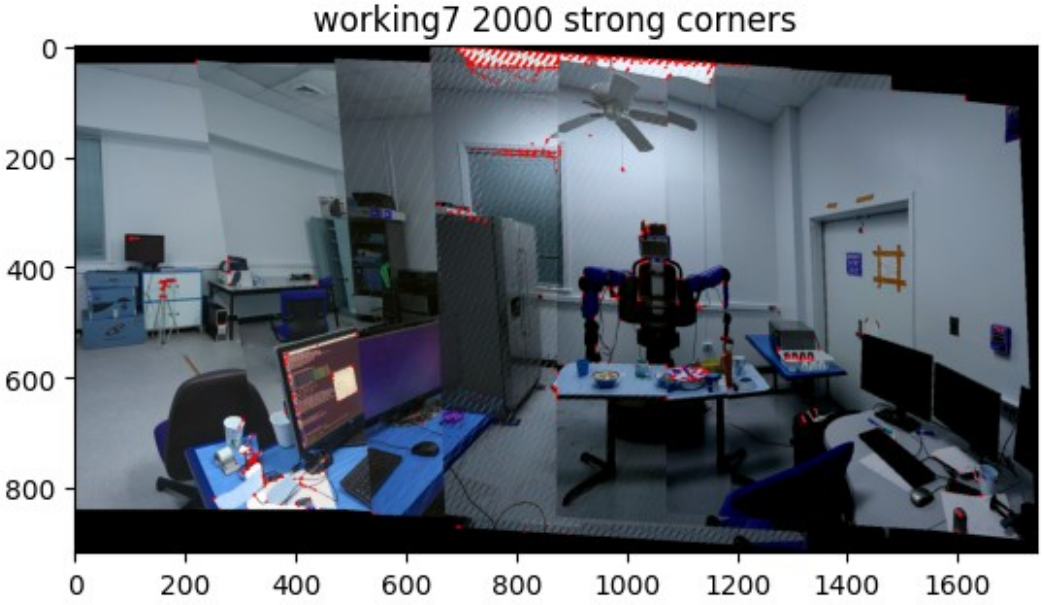

```
returning matrix with 13 inliers and mean error 501.5865621070088,
ranout of iterations
(13, 4)
```

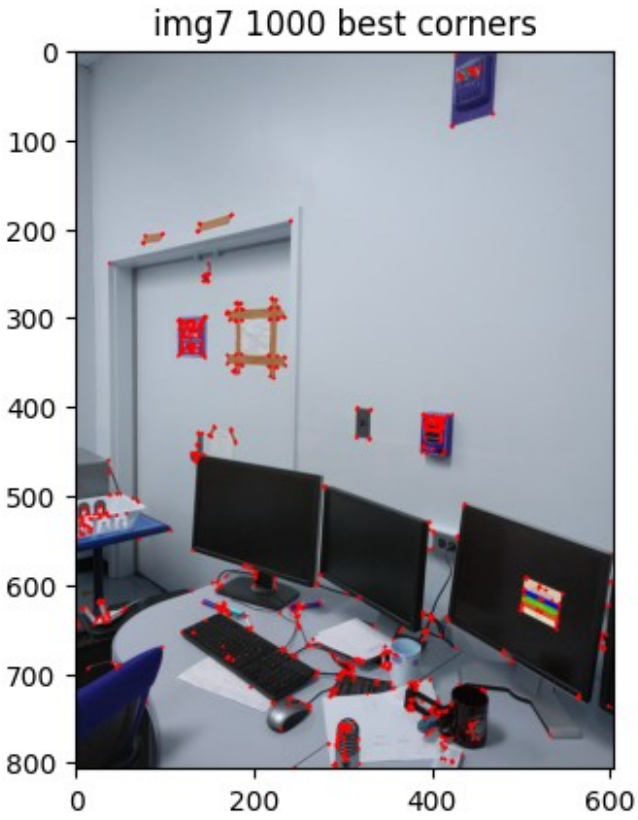


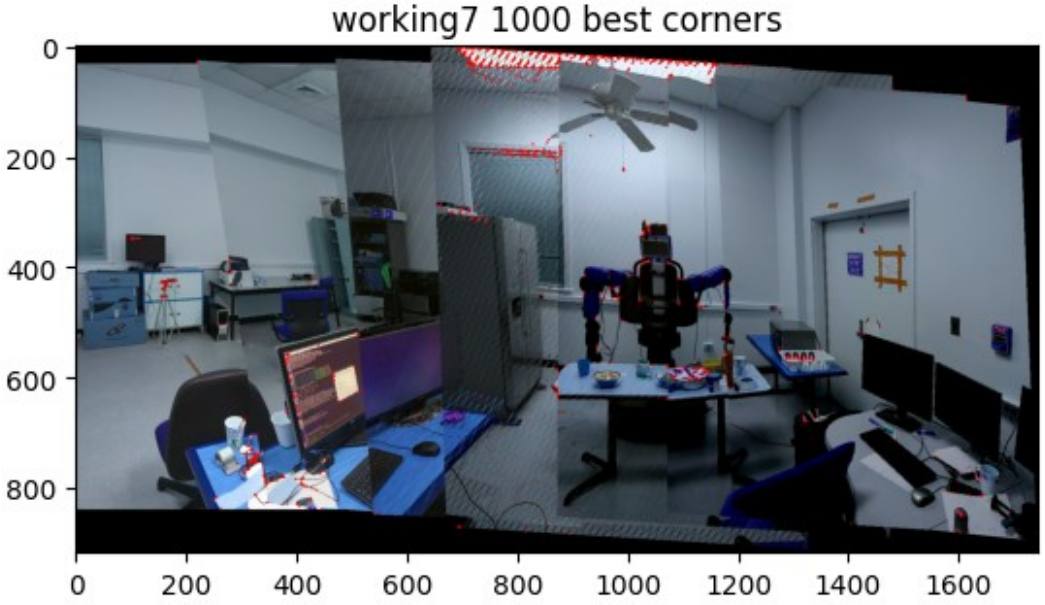

both imgs
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

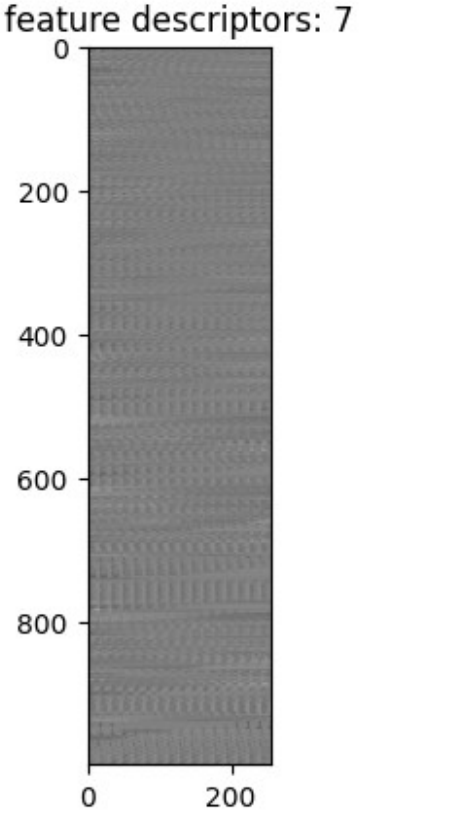

working img 7
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

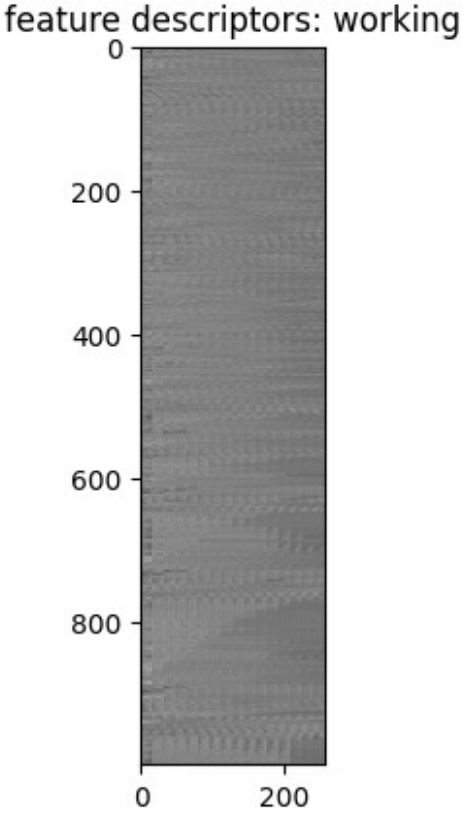

img7 2000 strong corners
0
100
200
300
400
500
600
700
800
0
200
400
600

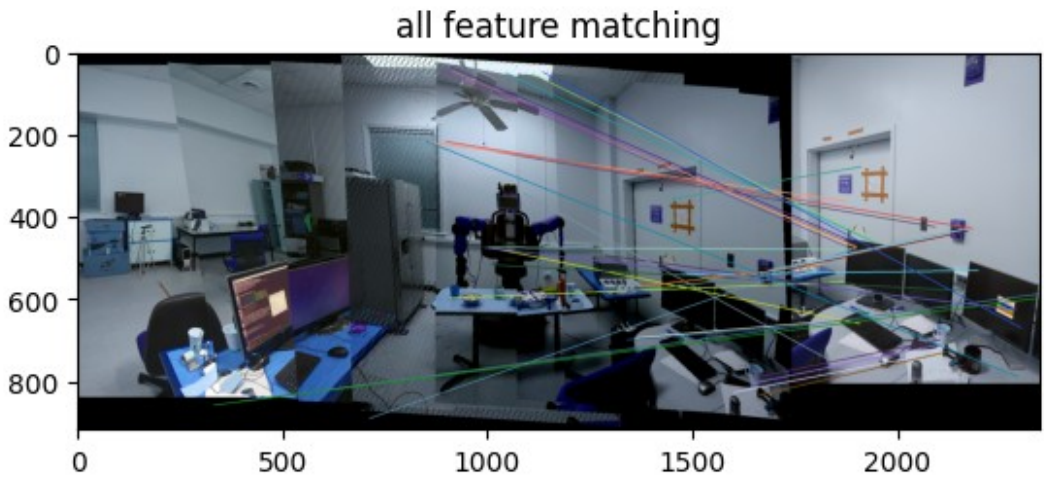

working7 2000 strong corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600


img7 1000 best corners
0
100
200
300
400
500
600
700
800
0
200
400
600

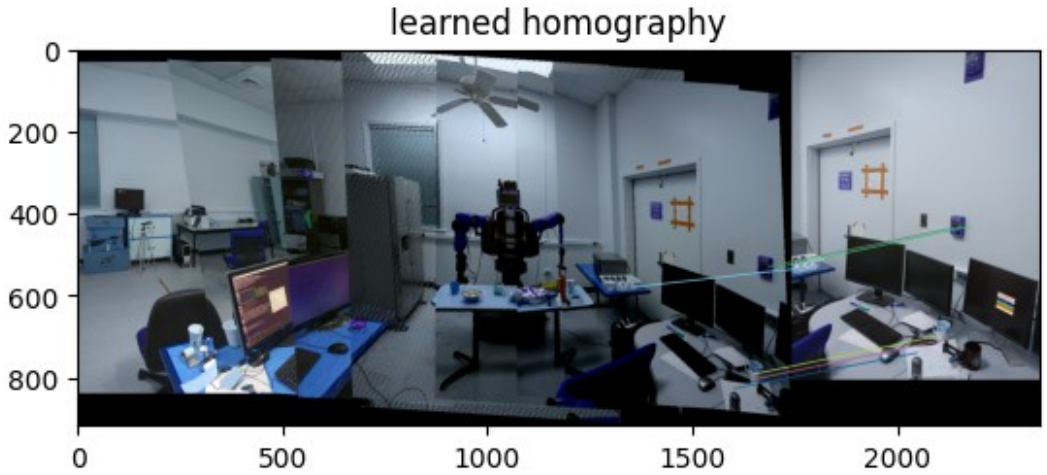

working7 1000 best corners
0
200
400
600
800
0
200
400
600
800
1000
1200
1400
1600

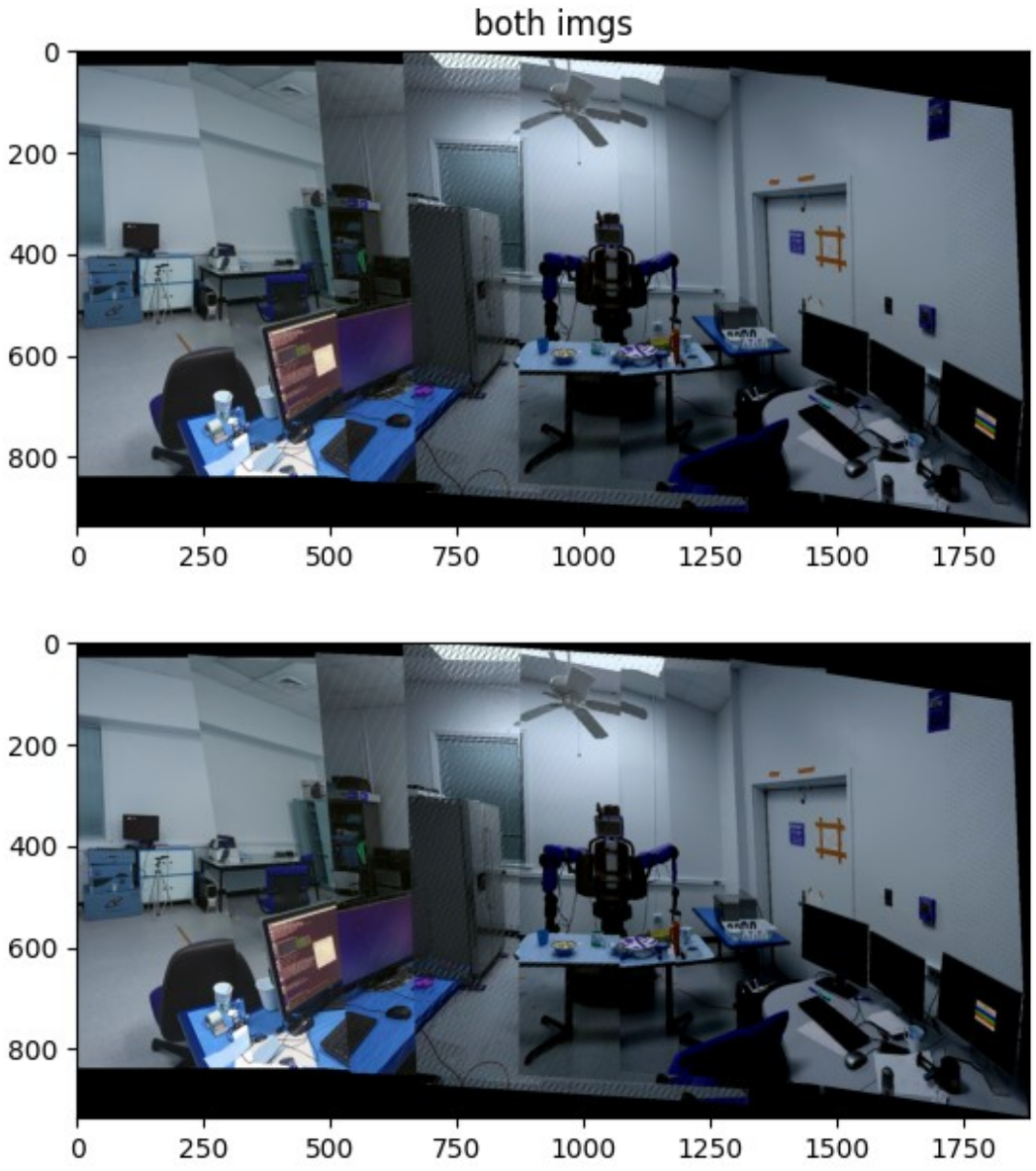

feature descriptors: 7
0
200
400
600
800
0
200


feature descriptors: working
0
200
400
600
800
0
200




```
returning matrix with 7 inliers and mean error 486.5574729310274,
ranout of iterations
(7, 4)
```

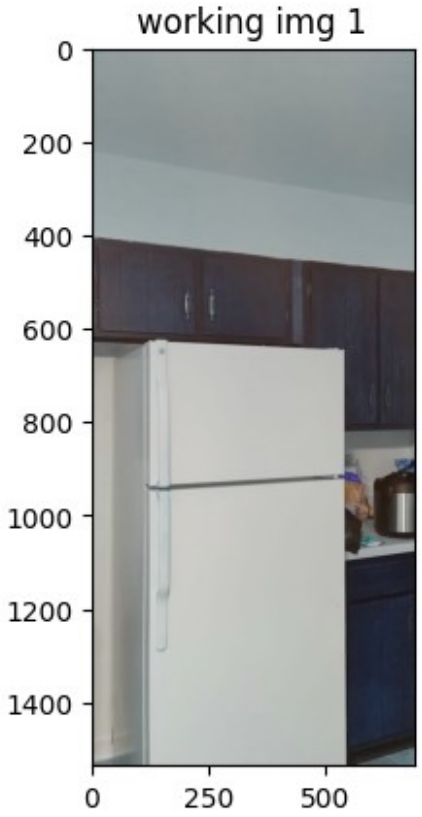


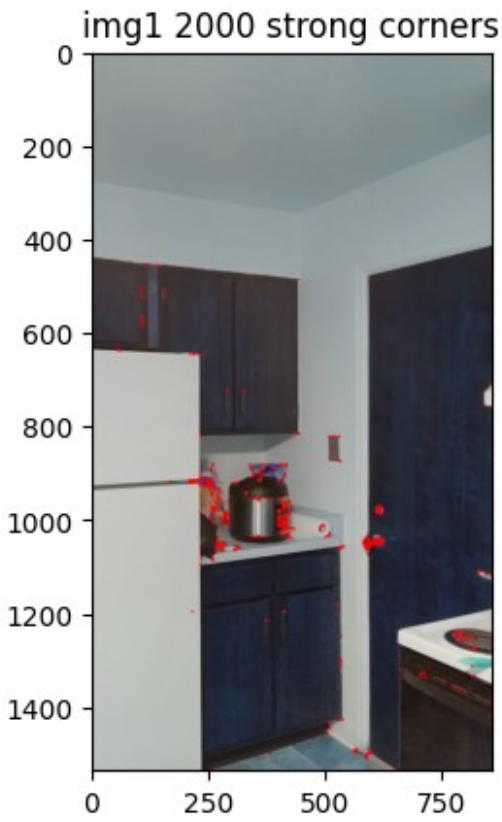

both imgs
0
200
400
600
800
0
250
500
750
1000
1250
1500
1750
0
200
400
600
800
0
250
500
750
1000
1250
1500
1750

Set 8 [5pts] (YourData)

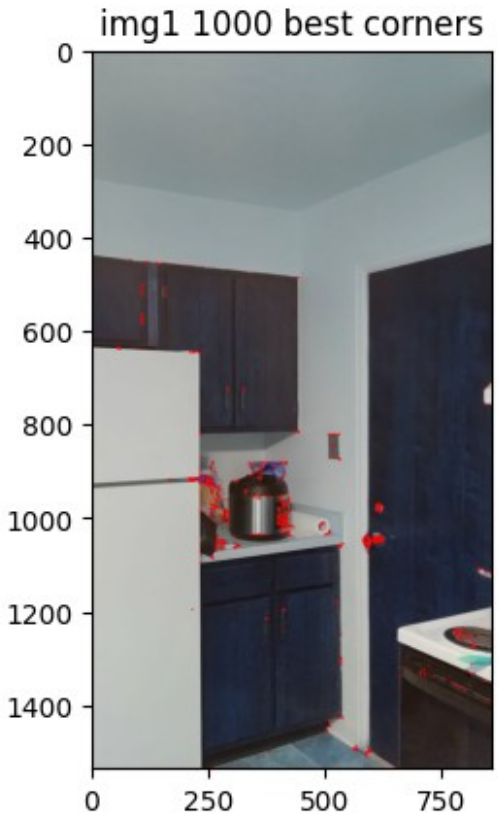
```
# YourData
# files =
gdown.download_folder(id="13zfICSDRsStGcIaf0CDbi1WqUZqQknah",
quiet=True, use_cookies=False)
files = ['/content/MCEN5228OurData/1.jpg',
'/content/MCEN5228OurData/2.jpg', '/content/MCEN5228OurData/3.jpg', ]

print(f"files: {files}")
img = mypano(files, num_strong_corners=2000, num_best_corners = 1000,
display = True)
show_img(img)
```

```
files: ['/content/MCEN5228OurData/1.jpg',
'/content/MCEN5228OurData/2.jpg', '/content/MCEN5228OurData/3.jpg']
```

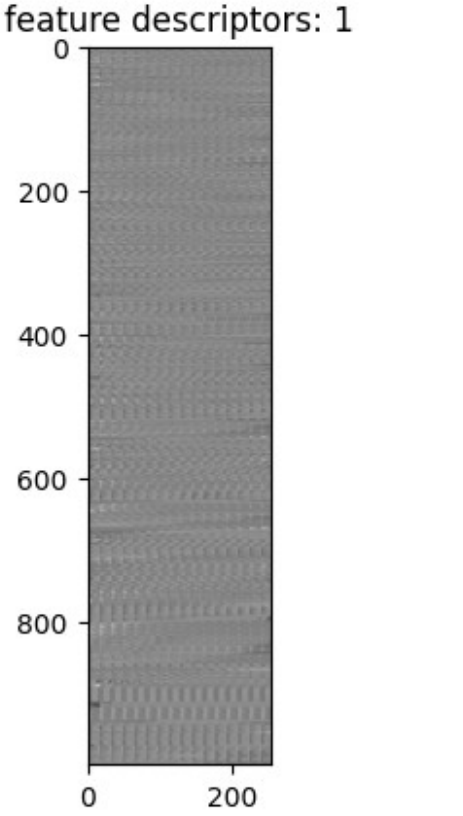


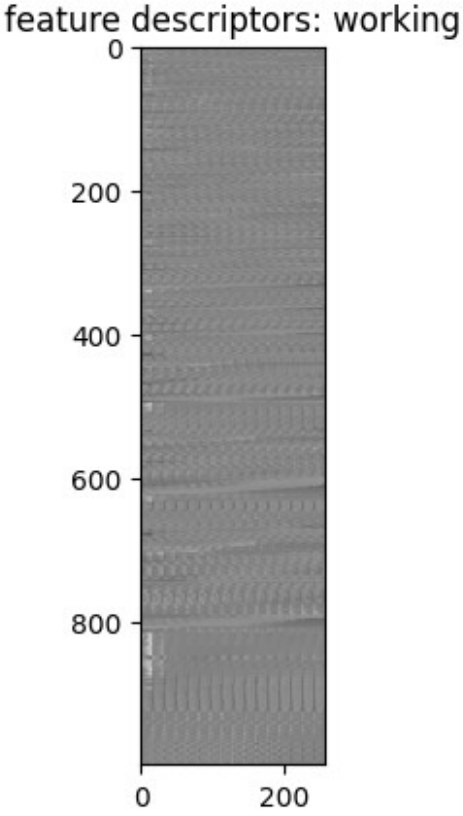

img1 2000 strong corners
0
200
400
600
800
1000
1200
1400
0
250
500
750

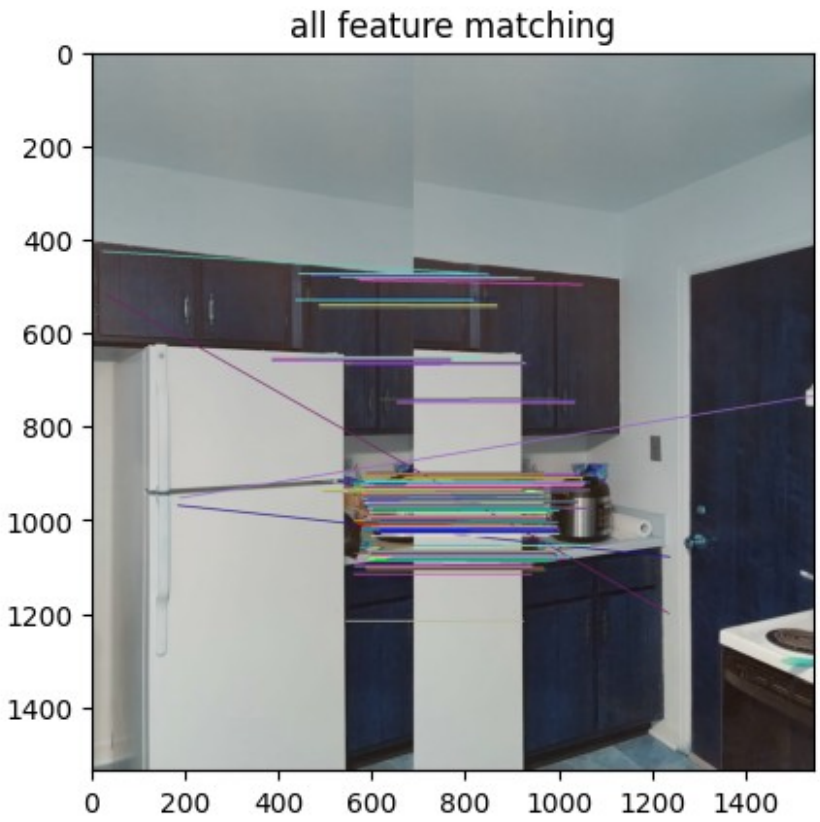

working1 2000 strong corners
0
200
400
600
800
1000
1200
1400
0
250
500


img1 1000 best corners
0
200
400
600
800
1000
1200
1400
0
250
500
750

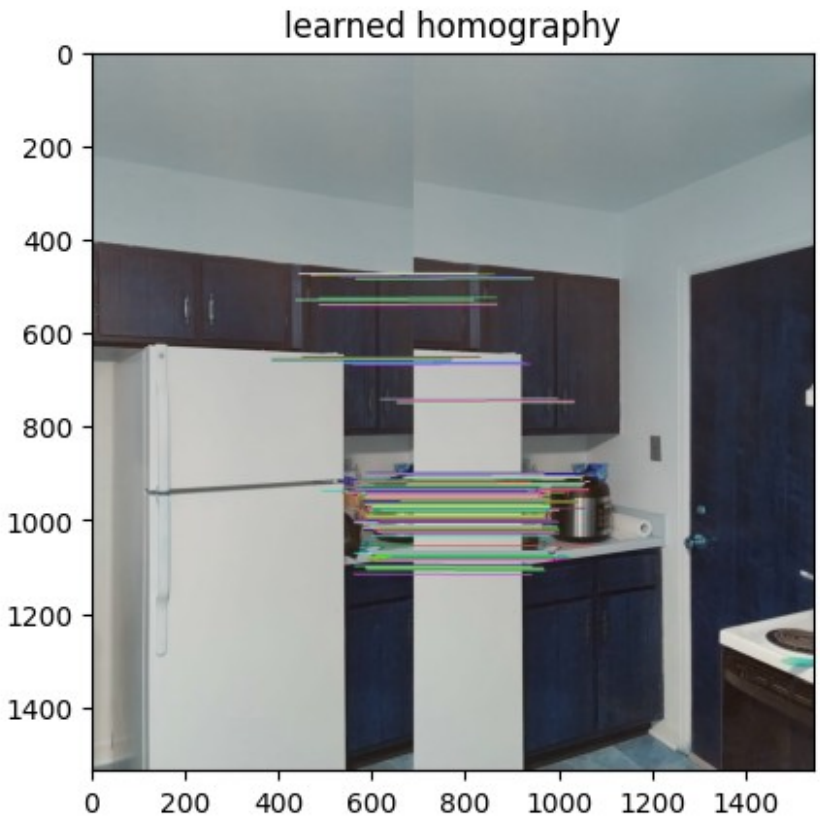

working1 1000 best corners
0
200
400
600
800
1000
1200
1400
0
250
500


feature descriptors: 1
0
200
400
600
800
0
200


feature descriptors: working
0
200
400
600
800
0
200

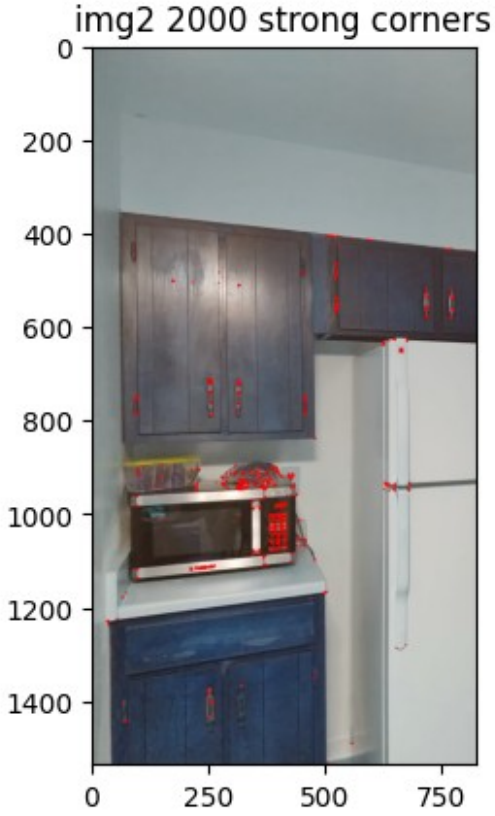



```
returning matrix because we have > 234.0 inliers, mean error:
13.957995633336067
(237, 4)
```


learned homography
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400


both imgs
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000

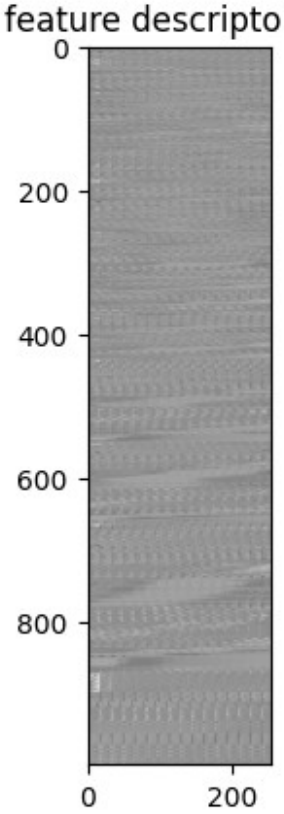

working img 2
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000

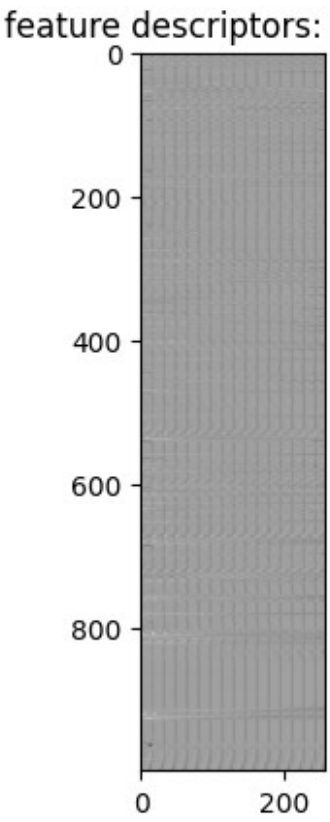

img2 2000 strong corners
0
200
400
600
800
1000
1200
1400
0
250
500
750

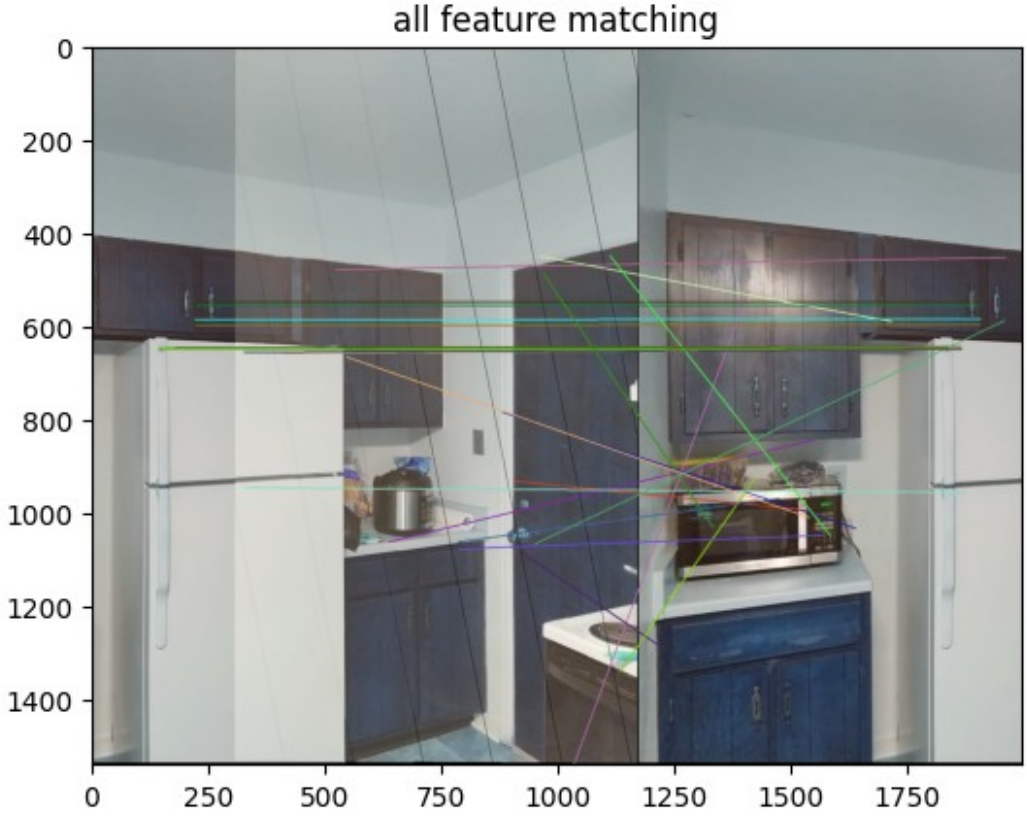

working2 2000 strong corners
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000


img2 1000 best corners
0
200
400
600
800
1000
1200
1400
0
250
500
750

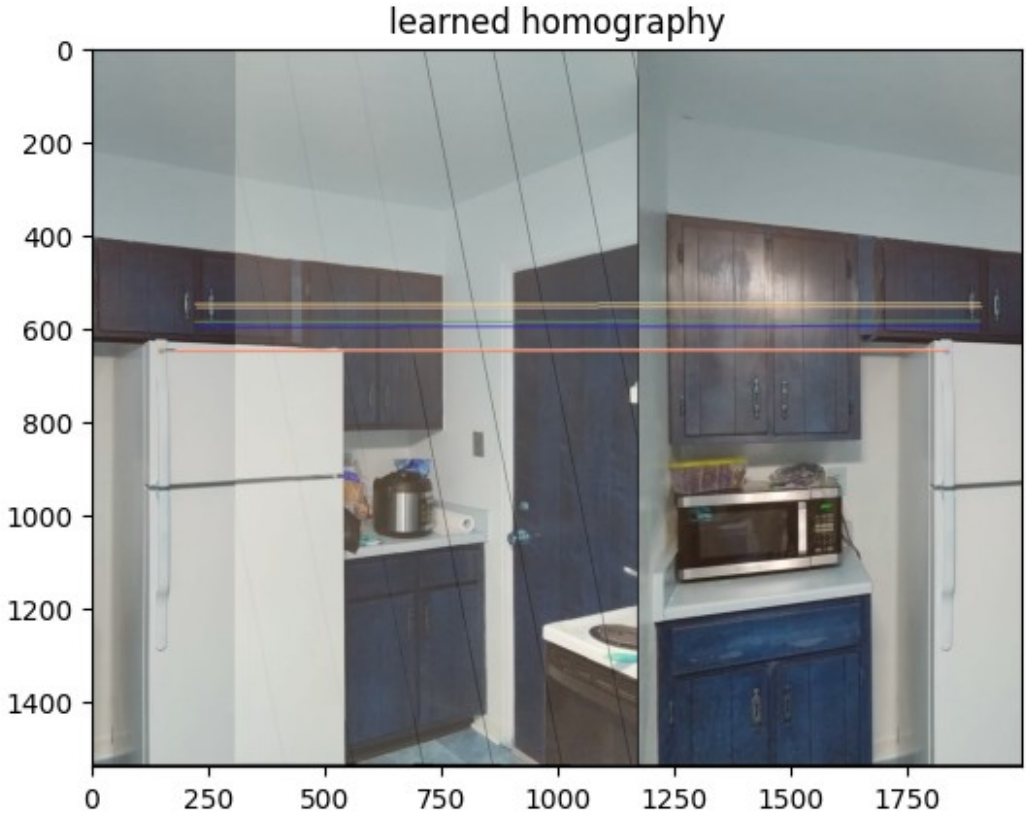

working2 1000 best corners
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000

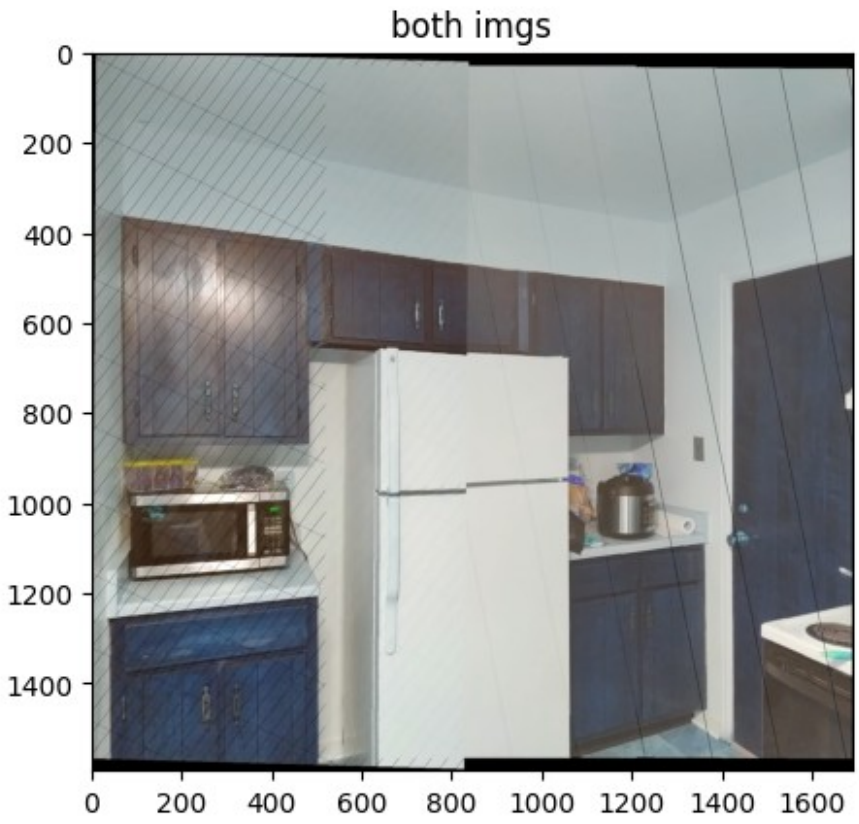

feature descriptors: 2
0
200
400
600
800
0
200

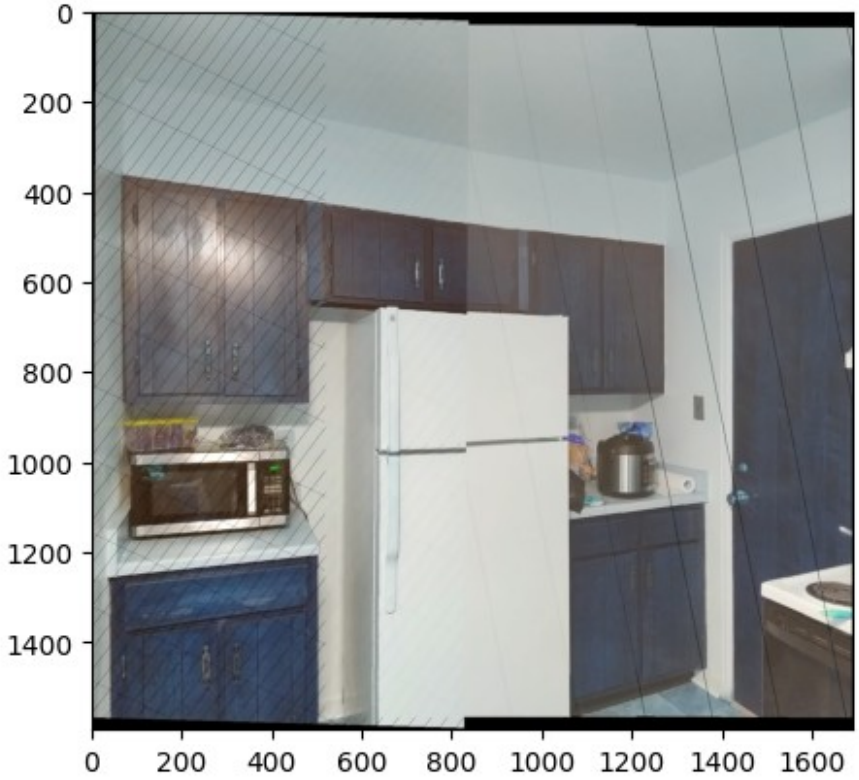

feature descriptors: working
0
200
400
600
800
0
200


all feature matching
0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500
1750

```
returning matrix with 7 inliers and mean error 636.3344271601931,
ranout of iterations
(7, 4)
```




both imgs
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400
1600


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400
1600