Homework_3_Report

December 2, 2023

1 Homework 3

Name: Syed Zain Raza

1.1 Problem 1: Feature Detection

1.1.1 Imports

[ ]:
```
# optional: allow Jupyter to "hot reload" the Python modules I wrote, to avoid␣
↪restarting the kernel after every change
%load_ext autoreload
%autoreload 2
```

[ ]:
```
import glob

import seaborn as sns
import matplotlib.pyplot as plt

import util
from util.corner_detection import HarrisCornerDetector
from util.ops import SimilarityMeasure

DATA_PATH = "./AlignmentTwoViews/*"
```

1.1.2 Part A: Harris Corner Detector

Image 1

[ ]:
```
img_paths = glob.glob(DATA_PATH)
```

[ ]:
```
img_paths
```

[ ]:
```
['./AlignmentTwoViews/uttower_right.jpg',
 './AlignmentTwoViews/uttower_left.jpg']
```

[ ]:
```
img1 = util.load_image(img_paths[0], return_array=True)
```

```
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 683 x 1024
```

[ ]:
```
corner_detector = HarrisCornerDetector()
response = corner_detector.detect_features(img1)
```

Let's just double check the properties of this `response` variable:

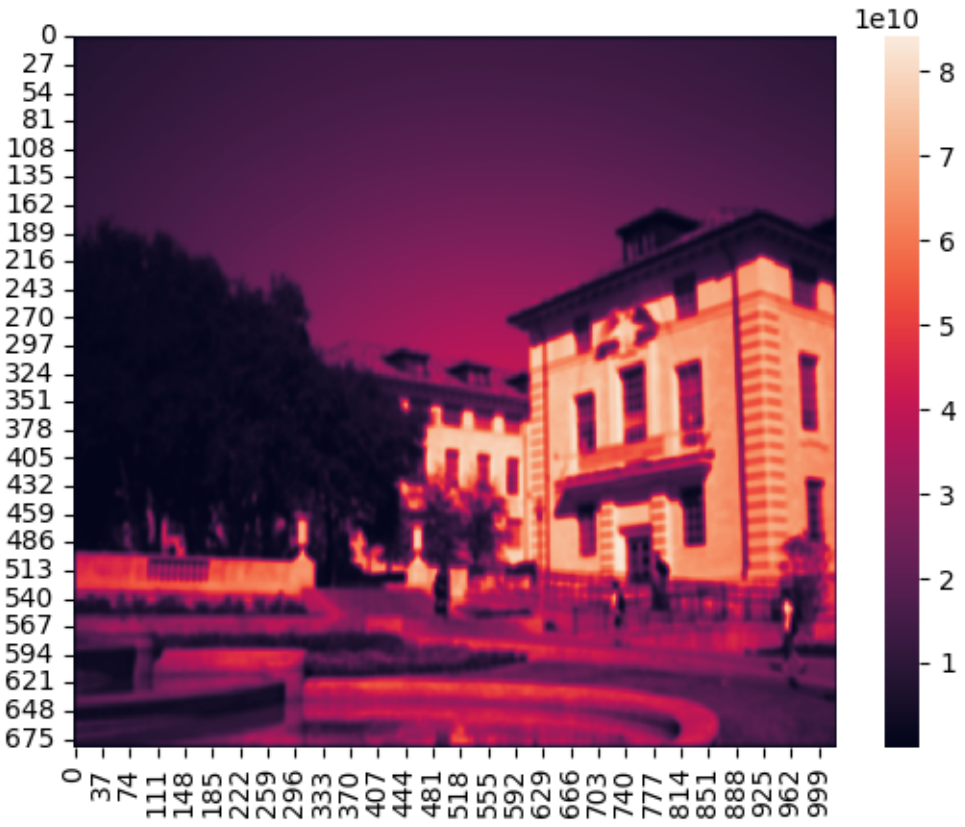
[ ]:
```
type(response)
```

[ ]:
```
numpy.ndarray
```

[ ]:
```
response.shape
```

[ ]:
```
(683, 1024)
```

[ ]:
```
sns.heatmap(response)  # just playing around, this is not my solution image
plt.show()
```



[ ]:
```
HarrisCornerDetector.execute_and_visualize(
    img1, "uttower_right",
    use_non_max_suppression=False,
)
```

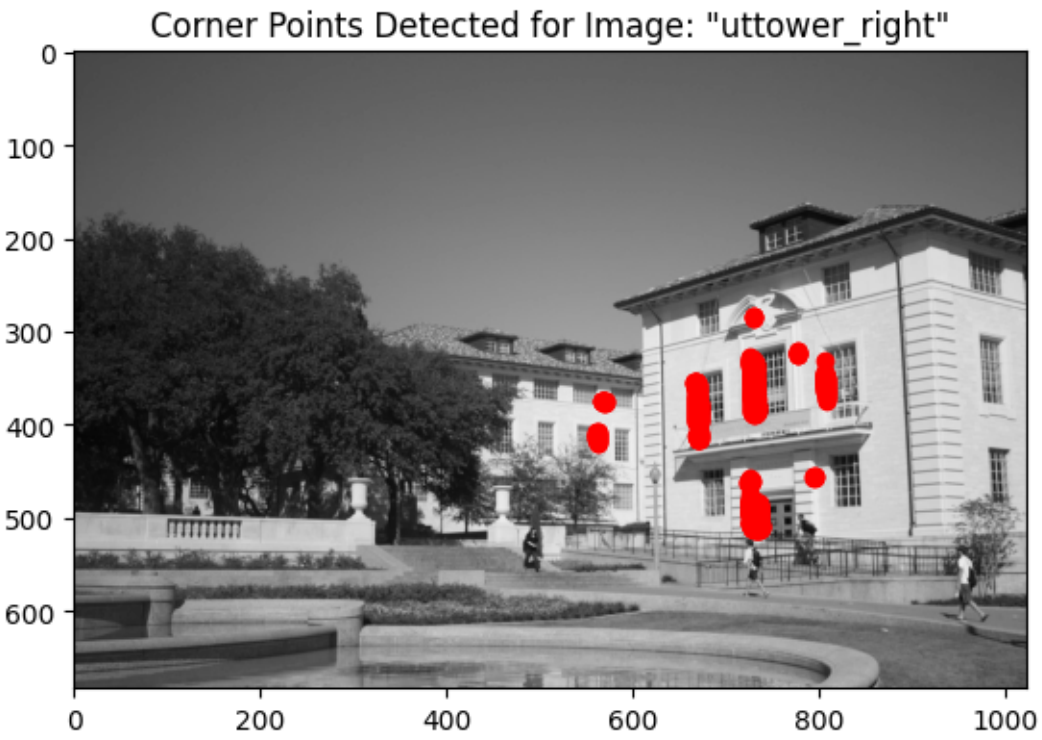
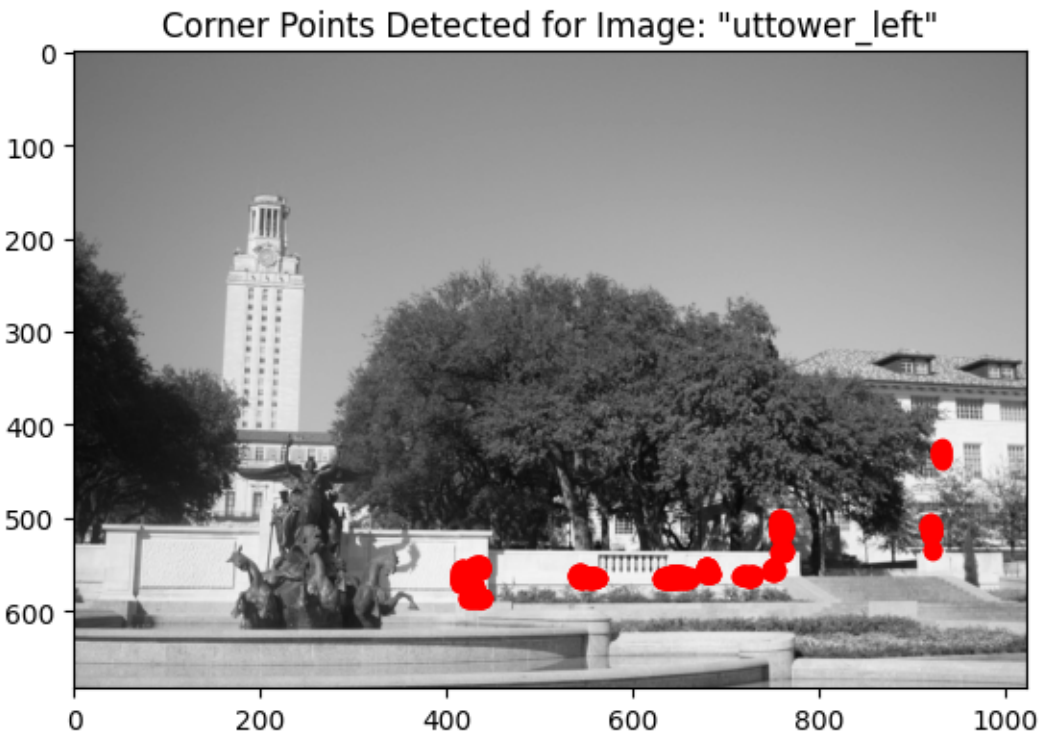


Image 2

[ ]:
```
img2 = util.load_image(img_paths[1], return_array=True)
```

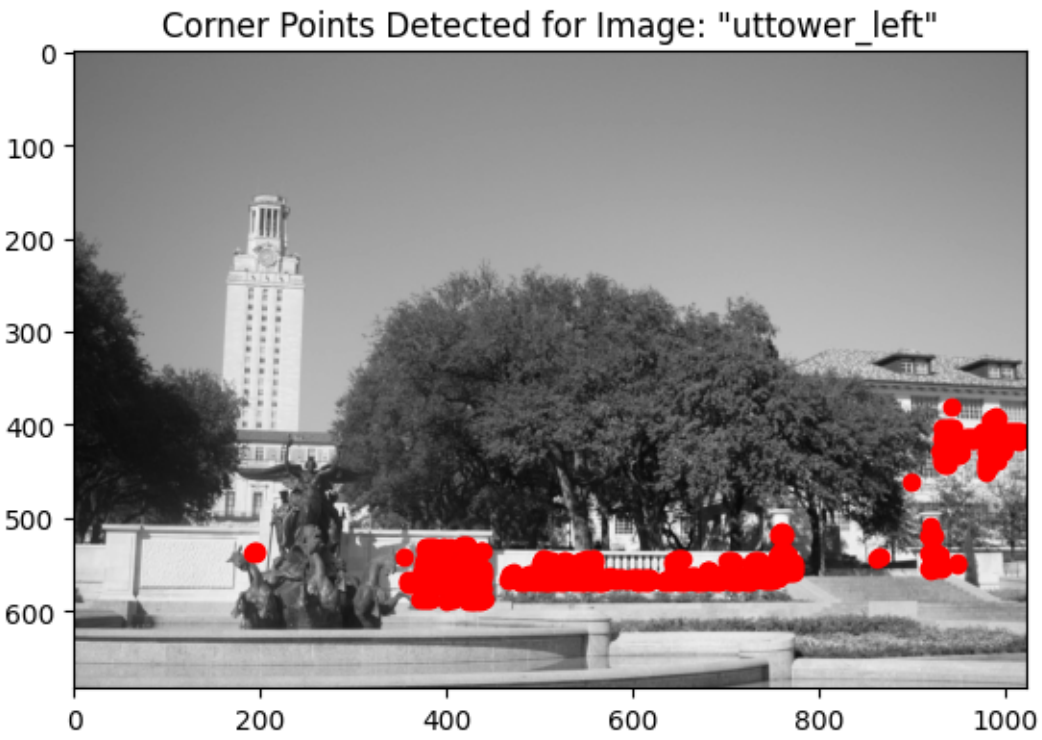
```
Dimensions of ./AlignmentTwoViews/uttower_left.jpg: 683 x 1024
```

[ ]:
```
HarrisCornerDetector.execute_and_visualize(
    img2,
    "uttower_left",
    use_non_max_suppression=False,
)
```

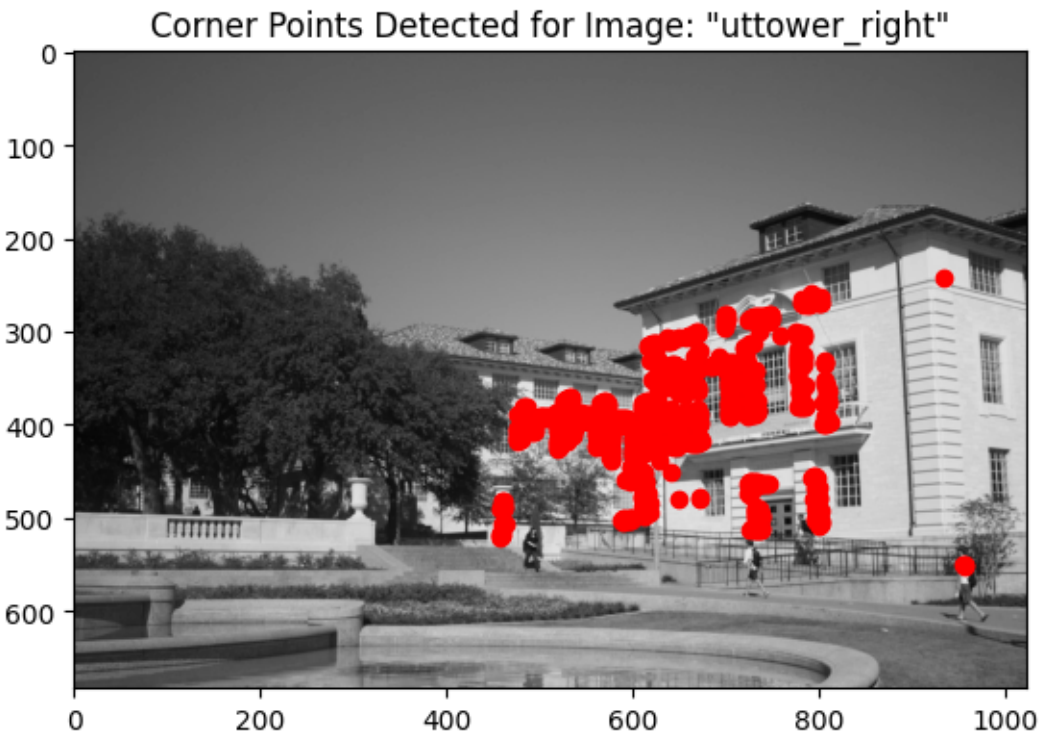


1.1.3 Part B: Leveraging Non-Maximum Suppression

[ ]:
```
HarrisCornerDetector.execute_and_visualize(
    img2,
    "uttower_left",
    use_non_max_suppression=True,
)
```



[ ]:
```
HarrisCornerDetector.execute_and_visualize(
    img1,
    "uttower_right",
    use_non_max_suppression=True,
)
```



1.1.4 Part C: Patch Similarity Measures

For convenience, let's recompute the features for both images (in case the cells above haven't been run):

[ ]:
```
corner_detector = HarrisCornerDetector()
```

[ ]:
```
img_paths = glob.glob(DATA_PATH)
right_img = util.load_image(img_paths[0], return_array=True)
left_img = util.load_image(img_paths[1], return_array=True)
```

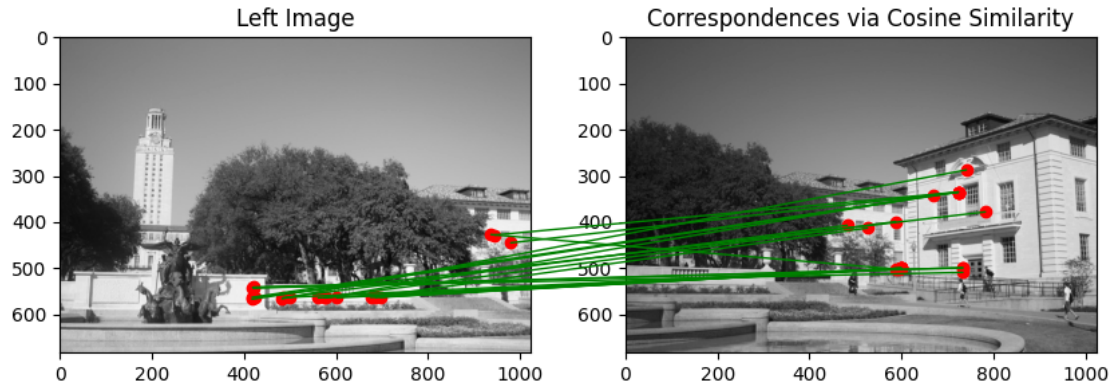
```
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 683 x 1024
Dimensions of ./AlignmentTwoViews/uttower_left.jpg: 683 x 1024
```

Similarity Measure: Cosine Similarity For this part, I'm choosing to go with the cosine similarity to measure the correspondence between different points in the two images.

Cosine similarity is a popular metric, and it is fast to compute (just like the SSD or NCC measures). But there are also other advantages that make it useful for this problem: - it is scale-invariant - and robust to outliers

Truthfully, for our use case these advantages may not entirely be required. But we'll use it nonetheless, as it's arguably a useful default.

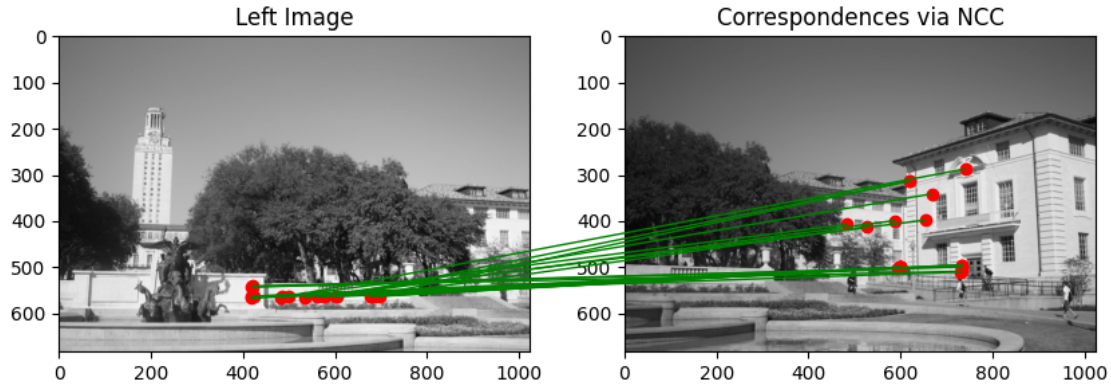
[ ]:
```
HarrisCornerDetector.find_and_visualize_correspondences(
    left_img,
    right_img,
    plot_title="Correspondences via Cosine Similarity",
    use_non_max_suppression=True,
    similarity_metric=SimilarityMeasure.COS,
)
```



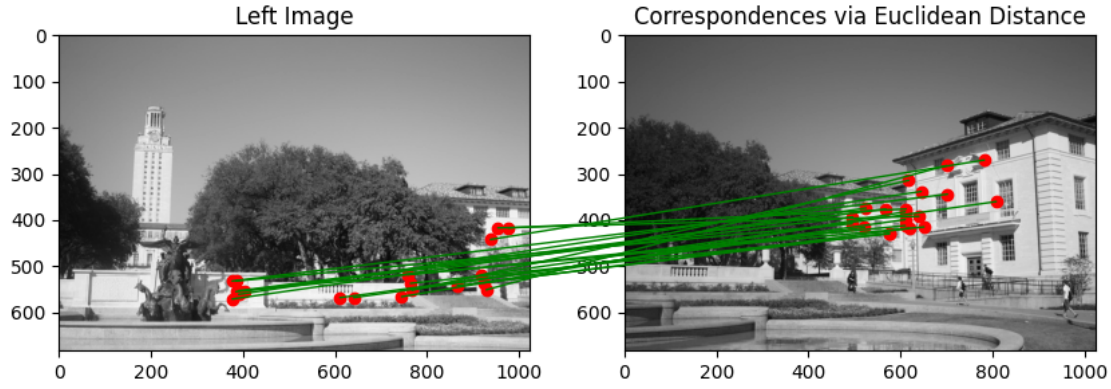
Discussion: SSD vs. NCC? As per the homework description, it's worth also exploring other similarity measures like the sum of squared differences (abbreviated SSD, aka the Euclidean distance) and normalized cross correlation (abbreviated NCC).

In general, there are some noteworthy differences between these two: - SSD represents the straight-line distance between two points in a given vector space, while NCC measures the linear relationship between functions (as measured by the distribution of their output values, which can be stored in vectors). - NCC is normalized, so it is better at being scale-variant (but that probably doesn't matter in our use case anyway). - SSD can be skewed more easily in the presence of outliers (but that shouldn't really matter here, as our measure is being computed with just two vectors). - as we can see below, the correspondences found when using NCC tend to concentrate more along one area of the left image. This might not indicate a shortcoming with the metric itself though - it could be an issue with my implementation, which zero-pads the patches (when they go outside the image bounds). Although this sounds like an acceptable idea in theory, in practice it is possible that it "dilutes" the similarity measure, which throws off our feature matching. - so overall, the correspondences found by SSD were the best in this case - indicating that for simple matching problems like ours, it is perfectly acceptable.

[ ]:
```
HarrisCornerDetector.find_and_visualize_correspondences(
    left_img,
    right_img,
    plot_title="Correspondences via NCC",
    use_non_max_suppression=True,
    similarity_metric=SimilarityMeasure.NCC,
)
```



[ ]:
```
HarrisCornerDetector.find_and_visualize_correspondences(
    left_img,
    right_img,
    plot_title="Correspondences via Euclidean Distance",
    use_non_max_suppression=True,
    similarity_metric=SimilarityMeasure.SSD,
)
```



1.1.5 Part D: Sensitivity to Rotation

In this next exploration, we should experiment with how SSD and NCC respond to rotations in the image.

Different Rotation Angles I will be rotating the right image. We'll use several rotation angles, for completeness:

[ ]:
```
# load the left image like normal
img_paths = glob.glob(DATA_PATH)
```

```
left_img = util.load_image(img_paths[1], return_array=True)
```

```
Dimensions of ./AlignmentTwoViews/uttower_left.jpg: 683 x 1024
```

[ ]:
```
rotation_angles = [0, 15, 30, 45, 60, 75]
```

[ ]:
```
right_images = [
    util.load_image(
        img_paths[0],
        rotation_angle=angle_rotation,
        return_array=True
    )
    for angle_rotation in rotation_angles
]
```

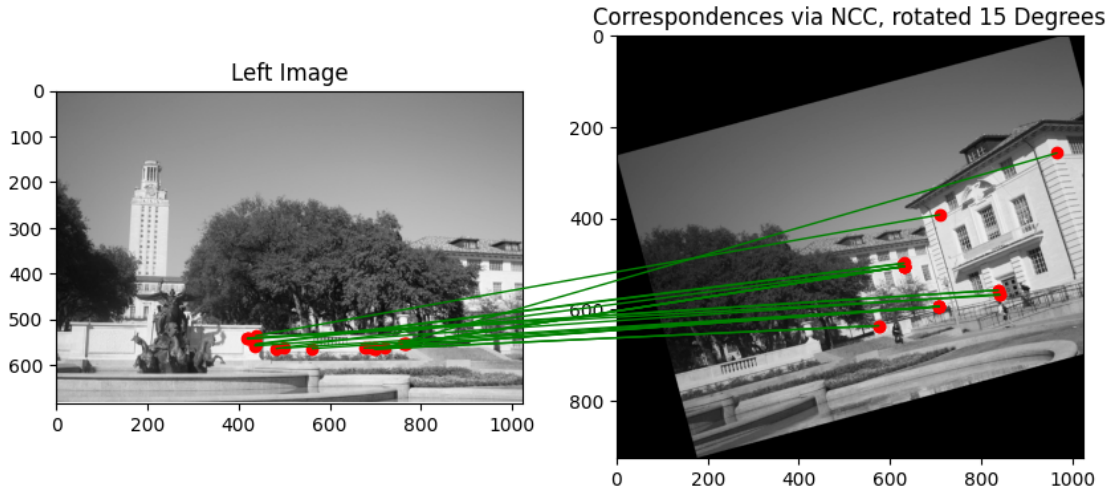
```
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 925 x 1166
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 1105 x 1230
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 1209 x 1208
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 1229 x 1104
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 1167 x 926
```

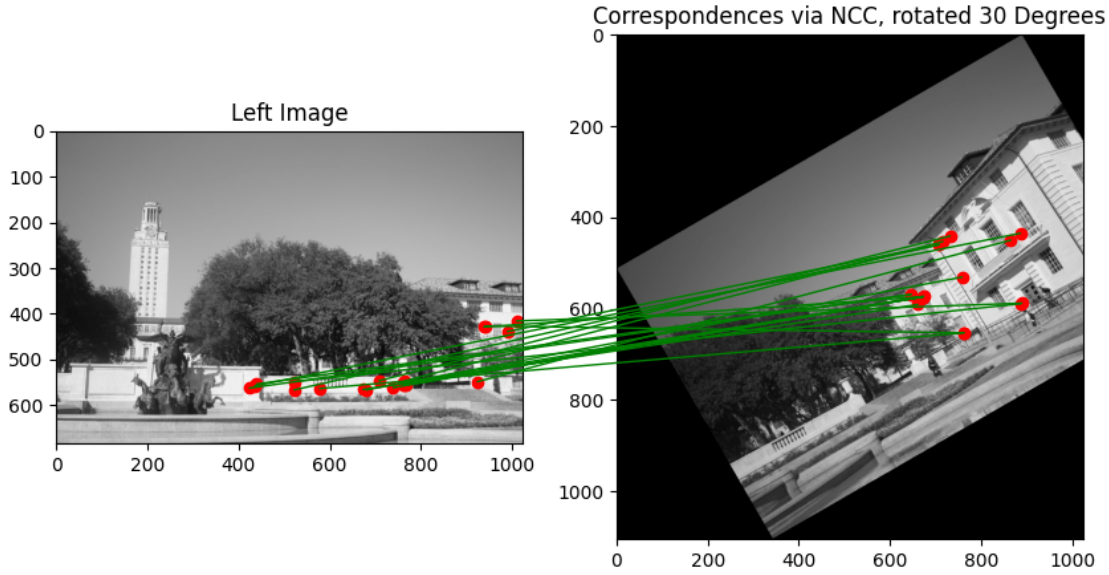
How does NCC Respond to Rotation?

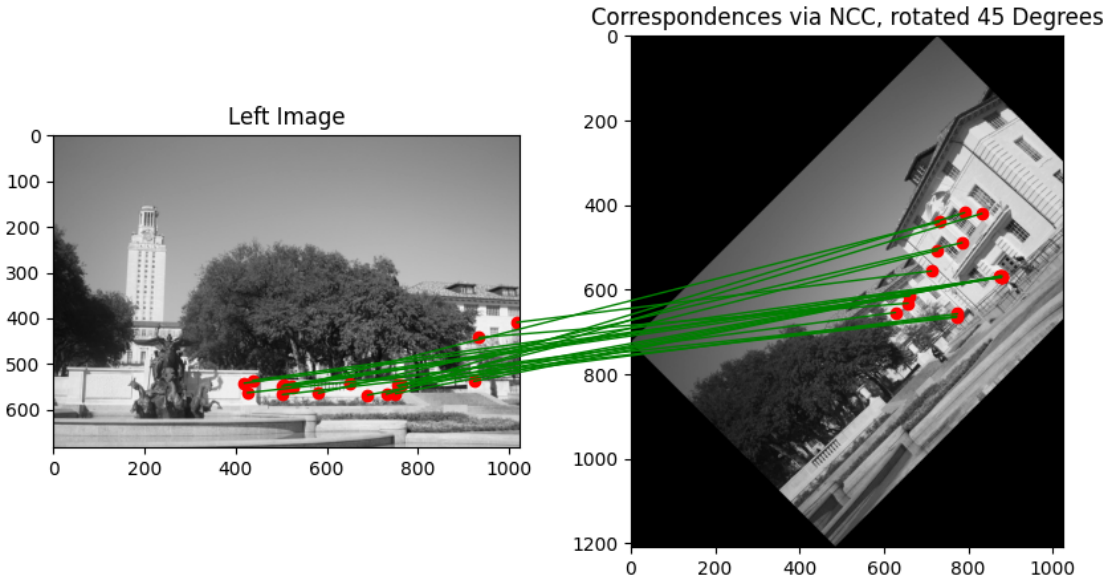
[ ]:
```
corner_detector = HarrisCornerDetector()
```

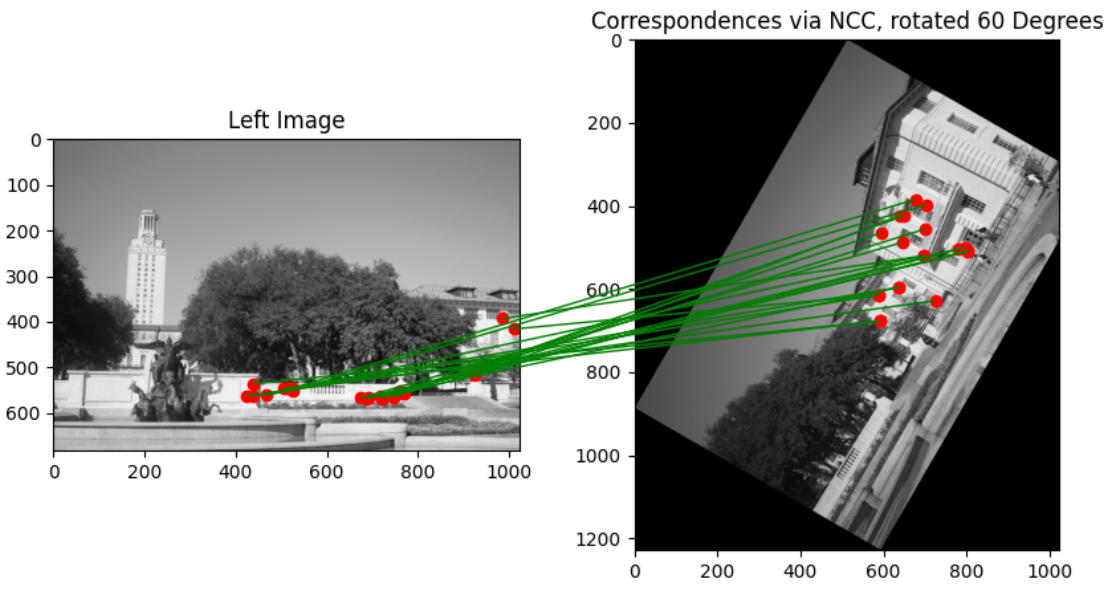
[ ]:
```
for index, rotation_degrees in enumerate(rotation_angles):
    HarrisCornerDetector.find_and_visualize_correspondences(
        left_img,
        right_images[index],
        plot_title=f"Correspondences via NCC, rotated {rotation_degrees}␣
↪Degrees",
        use_non_max_suppression=True,
        similarity_metric=SimilarityMeasure.NCC,
    )
```

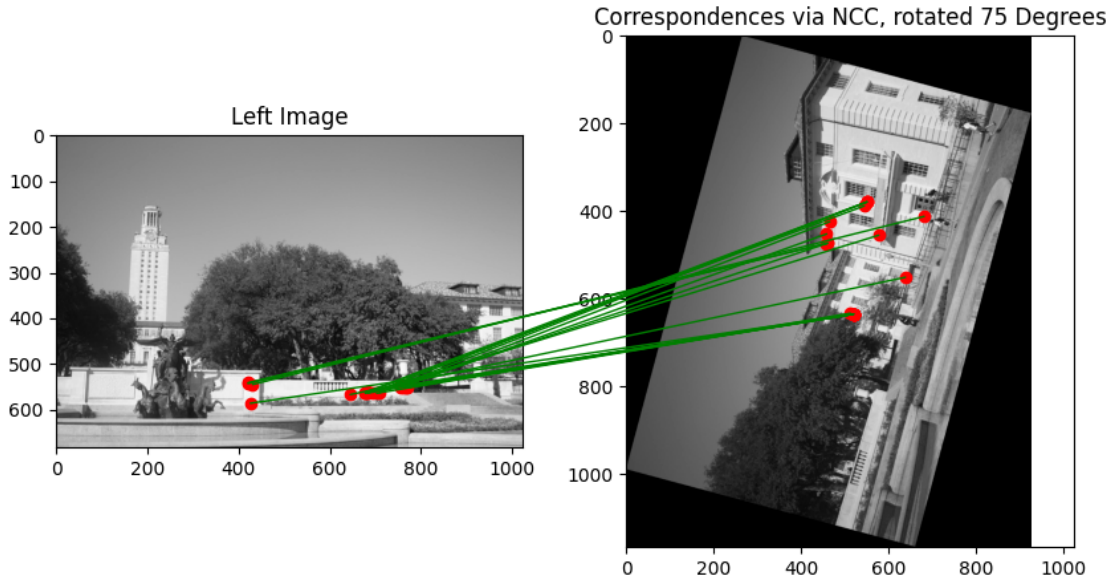
```
/Users/zainraza/Downloads/dev/courses/Stevens/CS-558/Corner-Detection-and-View-
Alignment/util/ops.py:48: RuntimeWarning: invalid value encountered in scalar
divide
  return numerator / denominator
```


Left Image
Correspondences via NCC, rotated 15 Degrees


Left Image
Correspondences via NCC, rotated 30 Degrees

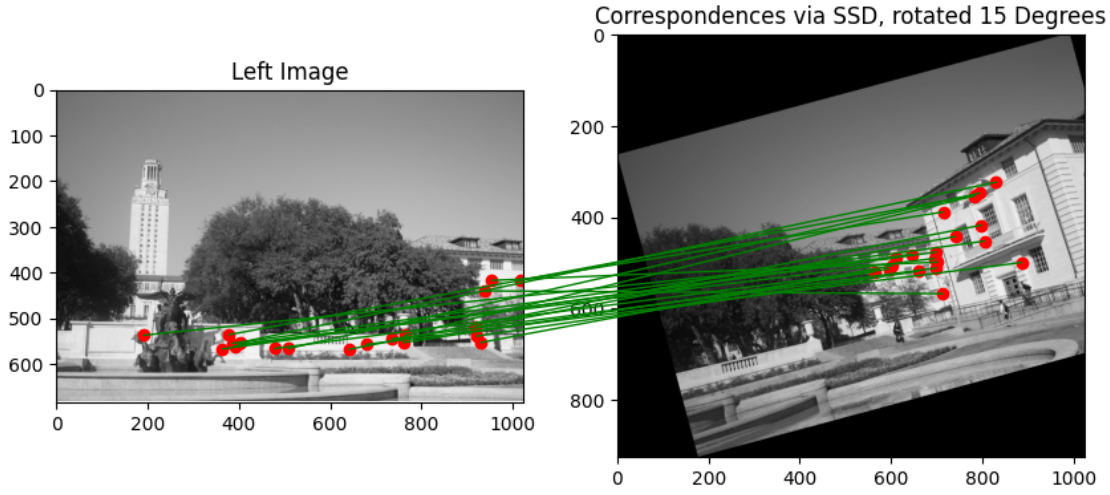

Left Image
Correspondences via NCC, rotated 45 Degrees

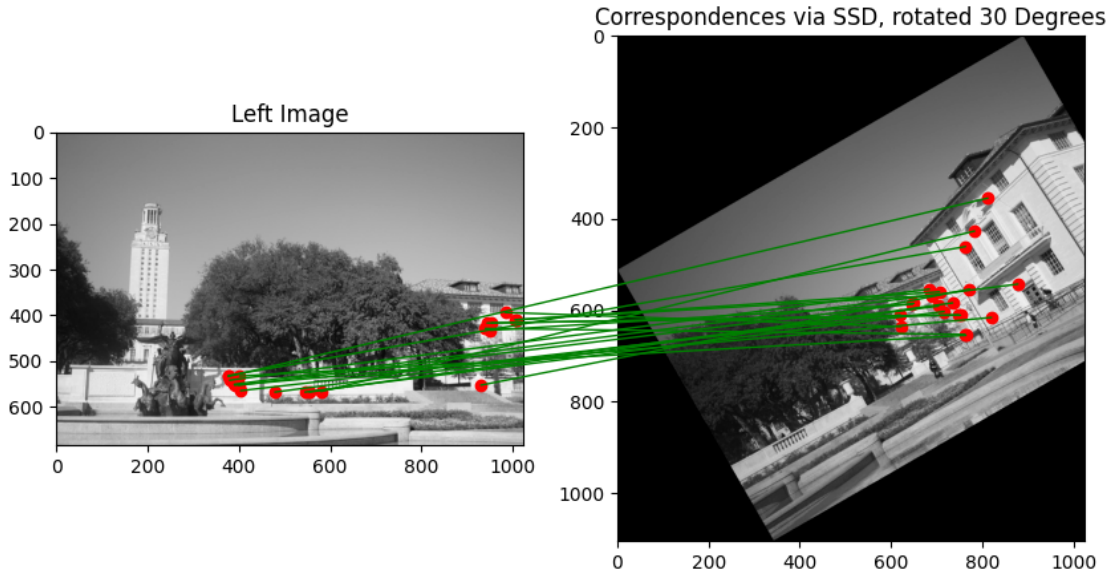

Left Image
Correspondences via NCC, rotated 60 Degrees

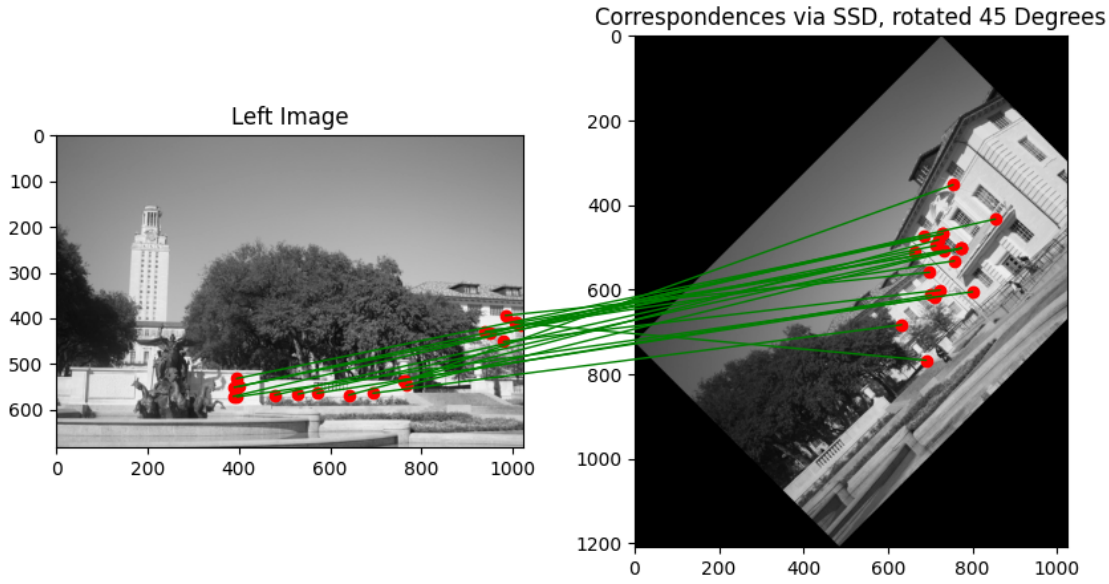


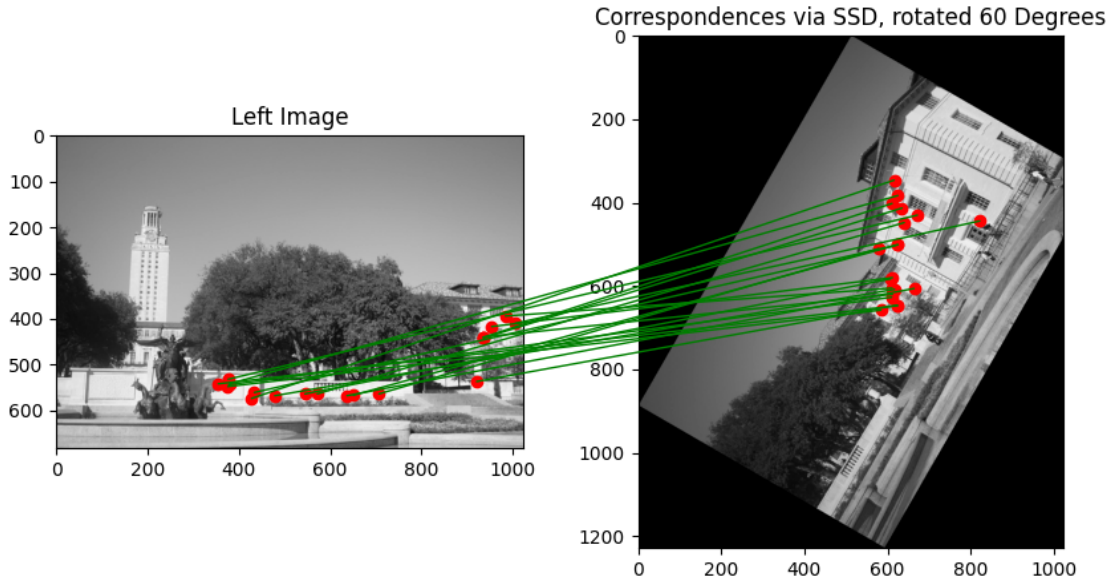
How does SSD Respond to Rotation?

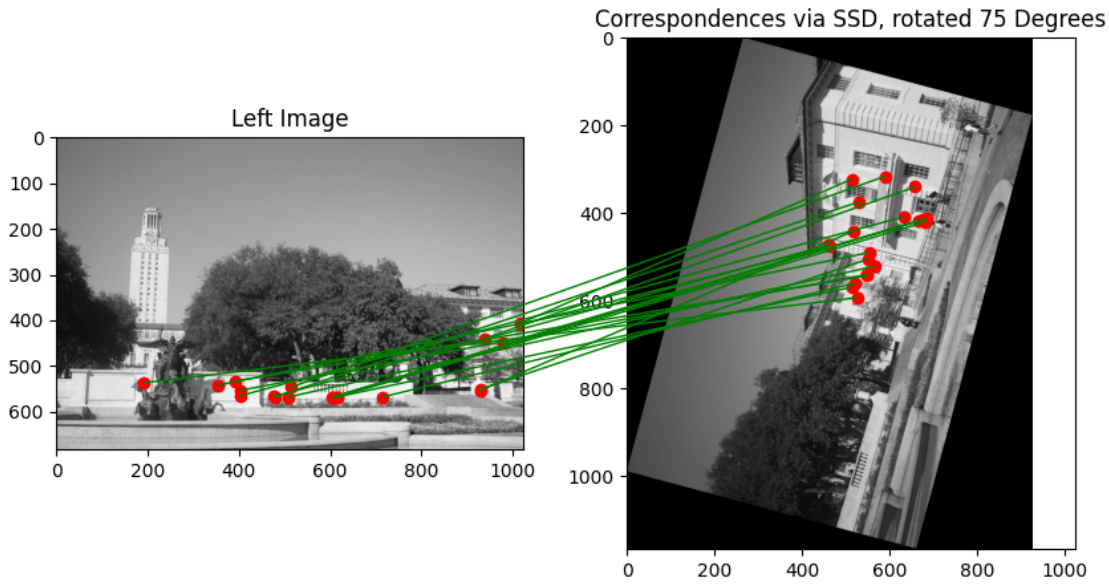
```
[ ]: for index, rotation_degrees in enumerate(rotation_angles):
         HarrisCornerDetector.find_and_visualize_correspondences(
             left_img,
             right_images[index],
             plot_title=f"Correspondences via SSD, rotated {rotation_degrees}␣
      ↪Degrees",
             use_non_max_suppression=True,
             similarity_metric=SimilarityMeasure.SSD,
         )
```




Left Image
Correspondences via SSD, rotated 30 Degrees


Left Image
Correspondences via SSD, rotated 45 Degrees





Discussion: Which Similarity Measure Responded Better? This answer relies on a qualitative observation as to which set of correspondences on the rotated images did a better job of accurately matching corners: those using the SSD similarity measure, or NCC.

To make this call, I will rate the quality of each set of correspondences on a scale from 1-3: - 1 = "poor" (e.g., if there are no correspondences between the "overlapping" part of the scene, i.e., the building that is shared between the two views) - 2 = "ok", which is when there are some correspondences between the areas of both images that overlap - 3 = "superb", which is when the

majority of the correspondences computed are accurate

Based on this system, we can make a total of how well each similarity measure performed, with respect to each rotation angle used. Then to declare the better measure, we propose to use the sum of the ratings:

Rotation (in degrees)	SSD	NCC
15	2	1
30	2	2
45	2	2
60	2	2
75	2	1
Total:	10	8

As we can see in the table, SSD scored a total of 10, while NCC only scored 8. This suggests SSD is slightly more robust to rotation than NCC.

Discussion: How to Be More Robust to Rotation? The crux of the answer here is not to actually focus on the similarity measure itself, but on the feature descriptors that it is being used to compare. In my implementation so far, I've been using a straightforward kind of descriptor - it's just a normalized 1D vector of the patch containing the corner point itself. The shortcoming of this approach is that it's not necessarily rotation invariant. A better approach to improve the robustness to robust would be to use a more sophisticated kind of descriptor, such as SIFT or SURF. In this way, it is possible that the correspondences computed by the similarity measure would rotation invariant.

1.2 Problem 2: Two View Image Alignment

[ ]:
```
# re-import as needed
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[ ]:
```
import glob

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

import util
from util.corner_detection import HarrisCornerDetector
from util.ops import SimilarityMeasure
from util.model_fitting import RANSACAffineTransformFitter

DATA_PATH = "./AlignmentTwoViews/*"
```

[ ]:
```
np.random.seed(42)  # for reproducibility
```

[ ]:
```
img_paths = glob.glob(DATA_PATH)
left_img = util.load_image(img_paths[1], return_array=True)
right_img = util.load_image(img_paths[0], return_array=True)
```

```
Dimensions of ./AlignmentTwoViews/uttower_left.jpg: 683 x 1024
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 683 x 1024
```

1.2.1 Part A: Building 2 Sets of Putative Features

To begin, we'll recompute two sets of feature correspondences - one based on highest similarity score (as determined by SSD), and the other will be a random selection.

[ ]:
```
# just a few configuration params
window_side_length = 3
use_non_max_suppression = True
top_many_features = 1000
```

The A1 Set: 20 Pairs Selected Based on Lowest SSD

[ ]:
```
detector = HarrisCornerDetector()
corner_response1 = detector.detect_features(left_img, use_non_max_suppression)
corner_response2 = detector.detect_features(right_img, use_non_max_suppression)
# pick top features
top_k_points1 = detector.pick_top_features(corner_response1, top_many_features)
top_k_points2 = detector.pick_top_features(corner_response2, top_many_features)
# compute feature descriptors, for top points
descriptors1 = HarrisCornerDetector.compute_feature_descriptors(
    left_img, top_k_points1, window_side_length
)
descriptors2 = HarrisCornerDetector.compute_feature_descriptors(
    right_img, top_k_points2, window_side_length
)

# Compute & plot the feature correspondences
a1_feature_correspondences = HarrisCornerDetector.
 ↪compute_feature_correspondences(
    descriptors=[descriptors1, descriptors2],
    desired_num_similarities=20,
    similarity_metric=SimilarityMeasure.SSD,
)
```

The A2 Set: 30 Randomly Selected Pairs

[ ]:
```
detector = HarrisCornerDetector()
corner_response1 = detector.detect_features(left_img, use_non_max_suppression)
corner_response2 = detector.detect_features(right_img, use_non_max_suppression)
# pick top features
top_k_points1 = detector.pick_top_features(corner_response1, top_many_features)
```

```
top_k_points2 = detector.pick_top_features(corner_response2, top_many_features)
# compute feature descriptors, for top points
descriptors1 = HarrisCornerDetector.compute_feature_descriptors(
    left_img, top_k_points1, window_side_length
)
descriptors2 = HarrisCornerDetector.compute_feature_descriptors(
    right_img, top_k_points2, window_side_length
)

# Compute & plot the feature correspondences
a2_feature_correspondences = HarrisCornerDetector.
 ↪compute_feature_correspondences(
    descriptors=[descriptors1, descriptors2],
    desired_num_similarities=30,
    similarity_metric=SimilarityMeasure.NULL,
)
```

[ ]:
```
a2_feature_correspondences.shape[0]
```

[ ]: 30

1.2.2 Part B: Estimating an Affine Transformation

B1: Using the A1 Set (Correspondences Chosen Based on Lowest SSD) We begin by running RANSAC once, without knowing what the expected number of iterations. This is so that we can get an initial set of outlier ratios. We'll use the average thereof to compute the expected iteration number. (And after that, we'll run RANSAC for real).

[ ]:
```
fitter = RANSACAffineTransformFitter()
```

[ ]:
```
model_results_b1, actual_num_iterations_b1, avg_e_b1 = fitter.fit(
    a1_feature_correspondences,
    distance_threshold=1000,  # random guess, b/c our descriptors led to poor␣
 ↪correspondences
    do_logging=False,
    prevent_resampling=False,
)
```

```
======= Iteration 1 Report: ========
No. of Inliers: 2
Outlier Ratio (e): 0.9
No. of Iterations (Expected): inf.
Avg reprojection error (1 iteration): 1759.5365720942577.
================================================
======= Iteration 2 Report: ========
No. of Inliers: 17
Outlier Ratio (e): 0.15000000000000002
No. of Iterations (Expected): 4602.867216938907.
```

```
Avg reprojection error (1 iteration): 675.55859335028.
================================================
======= Iteration 3 Report: ========
No. of Inliers: 9
Outlier Ratio (e): 0.15000000000000002
No. of Iterations (Expected): 4.836135322519713.
Avg reprojection error (1 iteration): 1298.108882491298.
================================================
======= Iteration 4 Report: ========
No. of Inliers: 2
Outlier Ratio (e): 0.15000000000000002
No. of Iterations (Expected): 4.836135322519713.
Avg reprojection error (1 iteration): 1527.7703800028862.
================================================
======= Iteration 5 Report: ========
No. of Inliers: 20
Outlier Ratio (e): 0.0
No. of Iterations (Expected): 4.836135322519713.
Avg reprojection error (1 iteration): 330.47115842005536.
================================================
No outliers… we have coverged! Stopping…
================= Global Results ===============
Average Outlier Ratio (e): 0.35000000000000003
Average reprojection error: 1118.2891172717552.
```

[ ]:
```python
expect_num_iter_b1 = RANSACAffineTransformFitter.compute_expected_num_iter(
    RANSACAffineTransformFitter.CONFIDENCE_LEVEL_FOR_NUM_ITERATIONS,  # 0.99
    e=avg_e_b1,
    s=3
)

print(f"Expected number of iterations for RANSAC: {expect_num_iter_b1}")
```

```
Expected number of iterations for RANSAC: 14.343352388783956
```

Ok, now let's do RANSAC again, for real:

[ ]:
```python
model_results_b1, actual_num_iterations_b1, avg_e_b1 = fitter.fit(
    a1_feature_correspondences,
    distance_threshold=1000,  # random guess, b/c our descriptors led to poor␣
 ↪correspondences
    do_logging=False,
    prevent_resampling=False,
    max_iter=expect_num_iter_b1
)
```

```
======= Iteration 1 Report: ========
No. of Inliers: 20
Outlier Ratio (e): 0.0
```

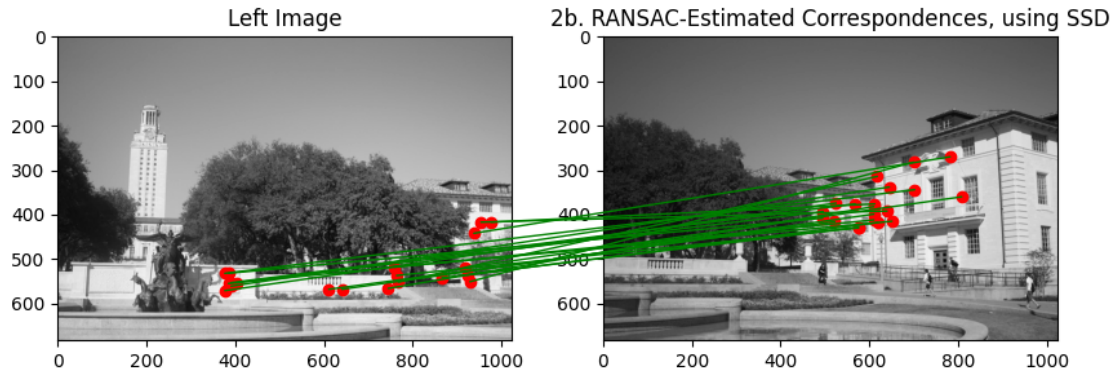
```
No. of Iterations (Expected): 14.343352388783956.
Avg reprojection error (1 iteration): 483.3908948960464.
================================================
No outliers… we have coverged! Stopping…
================= Global Results ===============
Average Outlier Ratio (e): 0.0
Average reprojection error: 483.3908948960464.
```

RANSAC Inlier Plot

[ ]:
```
best_model_found_b1 = model_results_b1[0]
best_model_params_b1 = best_model_found_b1[1]
best_model_params_b1
```

[ ]:
```
(array([[-0.20117616, -0.26700606],
        [-0.04932854, -0.53562714]]),
 array([[919.8906346 ],
        [647.55049592]]))
```

[ ]:
```
best_model_correspondences = best_model_found_b1[0]
HarrisCornerDetector.visualize_correspondences(
    left_img, right_img,
    best_model_correspondences,
    plot_title="2b. RANSAC-Estimated Correspondences, using SSD")
```



Discussion

- Regarding the expected number of RANSAC iterations for this experiment:
  - I set the inlier distance threshold to 1000 rather arbitrarily, after experimenting with values that were too small (i.e., `[3, 30, 300]`).
  - Based on that, the expected number of iterations for RANSAC would be approximately `14.34` (based on the first "mock" run of RANSAC, which had an average reprojection error of approximately `1118.29`).
  - Due to the distance threshold we set and the random sampling nature of RANSAC, the actual number of iterations was `1`.

B2: Using the A1 AND A2 Sets We will follow the same process as before - first, the mock RANSAC run:

[ ]:
```
fitter = RANSACAffineTransformFitter()

model_result_b2, actual_num_iterations, avg_e_b2 = fitter.fit(
    a2_feature_correspondences,
    distance_threshold=1000,
    do_logging=False,
)
```

```
======= Iteration 1 Report: ========
No. of Inliers: 5
Outlier Ratio (e): 0.8333333333333334
No. of Iterations (Expected): inf.
Avg reprojection error (1 iteration): 1881.7440979865435.
================================================
======= Iteration 2 Report: ========
No. of Inliers: 5
Outlier Ratio (e): 0.8333333333333334
No. of Iterations (Expected): 992.4123942696027.
Avg reprojection error (1 iteration): 1922.019024847616.
================================================
======= Iteration 3 Report: ========
No. of Inliers: 0
Outlier Ratio (e): 0.8333333333333334
No. of Iterations (Expected): 992.4123942696027.
Avg reprojection error (1 iteration): 2428.2190455659006.
================================================
======= Iteration 4 Report: ========
No. of Inliers: 0
Outlier Ratio (e): 0.8333333333333334
No. of Iterations (Expected): 992.4123942696027.
Avg reprojection error (1 iteration): 3947.4351071323.
================================================
======= Iteration 5 Report: ========
No. of Inliers: 10
Outlier Ratio (e): 0.6666666666666667
No. of Iterations (Expected): 992.4123942696027.
Avg reprojection error (1 iteration): 1314.8592241042302.
================================================
======= Iteration 6 Report: ========
No. of Inliers: 10
Outlier Ratio (e): 0.6666666666666667
No. of Iterations (Expected): 122.0225268863892.
Avg reprojection error (1 iteration): 500.8474769767474.
================================================
================ Global Results ===============
```

```
Average Outlier Ratio (e): 0.75
Average reprojection error: 2106.572326932822.
```

[ ]:
```python
expect_num_iter_b2 = RANSACAffineTransformFitter.compute_expected_num_iter(
    RANSACAffineTransformFitter.CONFIDENCE_LEVEL_FOR_NUM_ITERATIONS,  # 0.99
    e=avg_e_b2,
    s=3,
)

print(f"Expected number of iterations for RANSAC: {expect_num_iter_b2}")
```

```
Expected number of iterations for RANSAC: 292.42226317989287
```

Then the real RANSAC run:

[ ]:
```python
model_results_b2, actual_num_iterations_b2, avg_e_b2 = fitter.fit(
    a1_feature_correspondences,
    distance_threshold=1000,  # random guess, b/c our descriptors led to poor␣
 ↪correspondences
    do_logging=False,
    prevent_resampling=False,
    max_iter=expect_num_iter_b2,
)
```

```
======= Iteration 1 Report: ========
No. of Inliers: 20
Outlier Ratio (e): 0.0
No. of Iterations (Expected): 292.42226317989287.
Avg reprojection error (1 iteration): 512.4694555511358.
================================================
No outliers… we have coverged! Stopping…
================= Global Results ===============
Average Outlier Ratio (e): 0.0
Average reprojection error: 512.4694555511358.
```

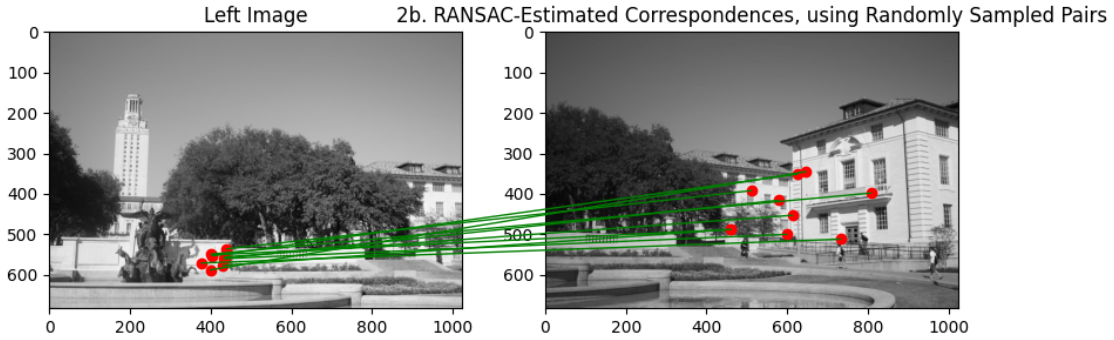
RANSAC Inlier Plot The inliers we found here are not as high-quality as in the last plot.

[ ]:
```python
best_model_found_b2 = model_result_b2[0]
best_model_params_b2 = best_model_found_b2[1]
best_model_params_b2
```

[ ]:
```
(array([[-1.49724518, -5.35703627],
        [ 1.21349862,  4.44427227]]),
 array([[ 4194.82535583],
        [-2598.14950643]]))
```

[ ]:
```python
best_model_correspondences = best_model_found_b2[0]
HarrisCornerDetector.visualize_correspondences(
    left_img,
```

```
    right_img,
    best_model_correspondences,
    plot_title="2b. RANSAC-Estimated Correspondences, using Randomly Sampled␣
 ↪Pairs",
)
```



Discussion

- Regarding the expected number of RANSAC iterations for this experiment:
  - I set the inlier distance threshold to 1000 rather arbitrarily, after experimenting with values that were too small (i.e., `[3, 30, 300]`).
  - Based on that, the expected number of iterations for RANSAC would be approximately `292.42` (based on the first "mock" run of RANSAC, which had an average reprojection error of approximately `2,106.57`).
  - Due to the distance threshold we set and the random sampling nature of RANSAC, the actual number of iterations was `1`, coincidentally.

1.2.3 Part C: Panorama Stitching

We begin by re-loading the images in color:

[ ]:
```
img_paths
```

[ ]:
```
['./AlignmentTwoViews/uttower_right.jpg',
 './AlignmentTwoViews/uttower_left.jpg']
```

[ ]:
```
right_img_color = util.load_image(
    img_paths[0],
    return_array=True,
    return_grayscale=False,
)
```

```
Dimensions of ./AlignmentTwoViews/uttower_right.jpg: 683 x 1024
```

[ ]:
```
left_img_color = util.load_image(
    img_paths[1],
    return_array=True,
    return_grayscale=False,
)
```

```
Dimensions of ./AlignmentTwoViews/uttower_left.jpg: 683 x 1024
```

[ ]:
```
from util.panorama_stitching import PanoramaStitcher
```

[ ]:
```
# because we're in RGB, our transform matrix/offset should be adjusted␣
 ↪accordingly
affine_transform_matrix = np.eye(3)
affine_transform_matrix[:2, :2] = best_model_params_b1[0]

affine_transform_offset = np.ones((3, 1))
affine_transform_offset[:2, 0] = best_model_params_b1[1].squeeze()

_ = PanoramaStitcher.composite_images(
    left_img,
    right_img,
    affine_transform_matrix_from_left_to_right=affine_transform_matrix,
    affine_transform_offset=affine_transform_offset,
    overlap_start_coordinate_x=300,  # just eye-balling where the overlap begins
    plot_title="Panorama of uttower",
)
```

Discussion The image above is supposed be a (transformed) version of the left image, composited with the right image. The left hand side of the panorama is black though, which suggests there is something wrong with the affine transformation parameters we are using. With more time, the likely next steps that could be taken to improve this panorama could be: 1) using better feature descriptors like SIFT, and a tighter distance threshold so that we could eventually find better parameters; and 2) we could also try processing the images in their RGB format (instead of

converting to grayscale). Both of these steps could possibly give us richer information to arrive at a higher-quality panorama.